**University of Alberta**

# Music-Driven Character Animation

by

Danielle Kristin Sauer

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Edmonton, Alberta
Spring 2007

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

*Your file Votre référence*
*ISBN: 978-0-494-30017-6*
*Our file Notre référence*
*ISBN: 978-0-494-30017-6*

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

---

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

# Abstract

Music-driven character animation extracts musical features from a song and uses them to create an animation. This thesis presents a system that builds a new animation directly from musical attributes, rather than simply synchronizing it to the music like similar systems. Using a simple script that identifies the movements involved in the performance and their timing, the user can control the animation of characters easily. Another unique feature of the system is its ability to incorporate multiple characters into the same animation, both with synchronized and unsynchronized movements. Two prototype systems are developed in this thesis: one incorporates hip-hop movement and the other integrates Celtic dance. An evaluation of the results from the Celtic system shows that the majority of animations are found to be appealing to viewers and that altering the music can change the attractiveness of the final result.

# Acknowledgments

First and foremost I would like to thank my fiancé, Steven Enns. He has been my rock for the last two years, providing endless encouragement and support. He dragged me away when I needed a break, consoled me when nothing was going my way, and cooked for me when I was too enthralled in my work to notice the time. I could not possibly have done this without him.

I owe another huge thank-you to my advisor, Herb Yang. He provided me with insight and inspiration and motivated me to achieve things I never thought possible. His dedication and enthusiasm towards my work made the past two years a fantastic experience. He was always available when I had a problem and his lightning fast e-mail response time was deeply appreciated.

I also want to thank the members of my group: Nathan Funk, Cheng Lei, Hai Mao, Daniel Neilson, Xuejie Qin, and Jason Selzer. Their input and feedback into my work solved many of my problems and prevented a few wrong turns along the way. Another person to whom I must give my gratitude is David Thue. I went to him for a lot of my problems with Maya, and even when he couldn't provide me with the solution, he somehow managed to help me figure it out myself.

Last, but certainly not least, I would like to thank my parents. They have been my role models throughout my life and have given me the skills necessary for completing this thesis. My dad has gifted me with his patience and my mom with her persistence, both of which are qualities that I needed these past two years. They continue to guide and encourage me, even when I don't realize that I need it.

# Table of Contents


**Chapter 1: Introduction** ..... **1**

**Chapter 2: Background and Related Work** ..... **4**
2.1 Music Analysis ..... 5
2.1.1 Signal Processing Techniques ..... 5
2.1.2 Tempo and Beat Detection ..... 9
2.2 Character Motion ..... 15
2.2.1 Keyframed Motion ..... 16
2.2.2 Motion-Capture Data ..... 17
2.2.3 Physical-Based Motion ..... 20
2.3 Synchronizing Music with Motion ..... 22
2.3.1 Synthesizing New Motion ..... 22
2.3.2 Direct Motion Editing ..... 24
2.3.3 Detecting Motion Features ..... 25
2.3.4 Keyframed Techniques ..... 26
2.4 Labanotation ..... 27
2.5 Motion Primitives ..... 28
2.6 Mapping ..... 29

**Chapter 3: Music Analysis** ..... **32**
3.1 Tempo Detection ..... 32
3.1.1 Algorithm Details ..... 32
3.1.2 Tempo Detection Results ..... 36
3.2 Beat Detection ..... 39
3.2.1 Original Algorithm Details ..... 40
3.2.2 New Algorithm Details ..... 41
3.2.3 Beat Detection Testing and Results ..... 45
3.3 Dynamics Extraction Algorithm ..... 51

**Chapter 4: Hip-Hop System** ..... **54**
4.1 System Overview ..... 55
4.2 Script Files ..... 56
4.2.1 Designing Script Files ..... 56
4.2.2 Parsing Script Files ..... 58
4.3 Mappings ..... 61
4.4 Primitive Movements ..... 63
4.5 Problems with the Hip-Hop System ..... 72

**Chapter 5: Celtic System** ..... **75**
5.1 System Overview ..... 76
5.2 Script Files ..... 77
5.2.1 Master Script File ..... 78
5.2.2 Secondary Script File ..... 80

5.2.3 Parsing Script Files ..... 85
5.3 Mappings ..... 87
5.3.1 Mapping Beats to Movement Timing ..... 88
5.3.2 Mapping Dynamics to Movement Distances ..... 89
5.4 Constraints ..... 90
5.5 Routines ..... 91
5.5.1 Built-in Routines ..... 91
5.5.2 User Designed Routines ..... 95
5.6 Primitive Movements ..... 96
5.7 Applications of Celtic System ..... 100

**Chapter 6: Results and Evaluation ..... 104**
6.1 Results ..... 104
6.2 Evaluation ..... 116
6.2.1 Evaluation Results ..... 117
6.2.2 Discussion ..... 119

**Chapter 7: Conclusion ..... 121**
7.1 Contributions ..... 121
7.2 Future Work ..... 122

**References ..... 124**

**A Animation Parameters ..... 128**

**B Evaluation Form ..... 130**

**C Script Files ..... 132**

# List of Tables

Table 3.1: Comparison of the old and new beat detection algorithms using a synthetic signal with a tempo of 82 bpm. The audio signal is 22 seconds in length, with a total of 31 beats. Testing is cut off at a noise level of ½ the beep's amplitude because this is where the new algorithm fails considerably. The second and third columns denote the number of beats out of 31 detected correctly by each algorithm. ..... 46

Table 3.2: The results of using the old beat detection algorithm on 10 synthetic signals with random seeds and a tempo of 153.3682 bpm. Eight different noise levels are used, ranging from 1/100$^{th}$ of the beep amplitude to ½ of the beep amplitude. ..... 48

Table 3.3: The results from performing beat detection with the new beat detection algorithm on ten synthetic signals with a tempo of 153.3682. Each signal was created with a different random seed and eight noise levels were used, ranging from 1/100 to ½ of the beep's amplitude. ..... 49

Table 6.1: Values that represent the time taken to build the script file, bake IK keys in Maya and render the entire animation for each song. ..... 116

Table 6.2: Overall results of the evaluation, taking into account the responses of all 18 people involved in the assessment of the animations. ..... 118

Table 6.3: Results of the evaluation split up by group into evaluators with dancing experience, evaluators with computer programming experience and evaluators with experience in neither. ..... 119

Table A.1: Lists the parameters that change for each animation. ..... 129

# List of Figures

Figure 2.1: The Fourier Transform takes a time-based signal (left) and converts it to a frequency-based signal (right). These diagrams were taken from [16]. ..... 6
Figure 2.2: The diagram on the left displays passing a signal through a high pass filter and a low pass filter. The result is two complementary subsignals and twice as much data as found in the original signal. The diagram on the right performs the same filtering operation, but downsamples the subsignals by 2. This still provides complete information about the original signal but reduces the number of samples by half. Both diagrams were taken from [30]..... 8
Figure 2.3: The Discrete Wavelet Transform (DWT) decomposes a signal into several frequency levels (as denoted by *f* at each level) by using highpass (g[n]) and lowpass filters (h[n]) and subsampling the results by 2. The length of the signal determines how many times the decomposition process can be performed. This diagram was taken from [31]. ..... 9
Figure 2.4: Scheirer's beat detection system tracks a song's tempo using a frequency filterbank and comb filter resonators. The frequency filterbank splits the signal into several frequency bands while the comb filters examine each band and search for a tempo that corresponds to the resonator's delay time. The diagram above was adopted from the original work. ..... 11
Figure 2.5: Using multiple agents to track beat hypotheses is a popular method in beat detection algorithms. In Dixon's algorithm, values A to F across the top represent the beat onsets, with the solid squares representing predicted beat times that occur on an onset and the hollow squares representing predicted beat times that don't correspond to onsets. Squares which occur close together (Agent1) correspond to a faster tempo than circles that occur further apart (Agent2). The diagram above was adopted from Dixon's original work. ..... 13
Figure 2.6: Using the previous beat and the estimated current beat interval, Jensen and Anderson's algorithm determines if the current note onset (the peak pointed to by the Beat location arrow) is the next beat in the audio signal. This diagram was adopted from Jensen and Anderson's original work. ..... 14
Figure 2.7: Pre-conditions, expected performance and post-conditions for the physics-based Fall controller designed by Faloutsos. The controller takes into account velocity, balance and environmental parameters such as ground contact. This controller diagram was redone based on Faloutsos' Fall controller. ..... 21
Figure 2.8: Example of Labanotation from the Dance Notation Bureau website [9]. ..... 27
Figure 3.1: An example of a beat histogram produced by the tempo detection algorithm. The values across the bottom represent the possible tempos. For this particular Celtic song, "Siamsa," the tempo is detected to be 117 beats per minute. ..... 37
Figure 3.2: A beat histogram for a 30 second Beatles song. The tempo is identified as 161 bpm when, in actuality, the tempo is 82 bpm. The

algorithm recognizes the faster beat as the tempo because it occurs more often in the signal. 38
Figure 3.3: The results of the new beat detection algorithm on a synthesized signal with a tempo of 82 bpm. The noise level is 1/2 and a 35% threshold value is used. Blue lines represent the original beats in the signal and red lines the beat positions detected by the algorithm. The top image displays the full signal after beat detection, while the bottom image a closer view of the distance between actual beats and detected ones. 47
Figure 3.4: The results of the new beat detection algorithm on a synthesized signal with a tempo of 153.3682 bpm. The noise level is 1/100. Blue lines represent the original beats in the signal and red lines represent the beat positions detected by the algorithm. 50
Figure 3.5: A signal graph of the Celtic song "Warriors" used by the Celtic animation system. The tempo of this song is 135 bpm. The blue lines symbolize the original signal and the red lines represent the detected beats. One can see a space where a beat should occur around the 65,000-sample position, as marked by the green square. The algorithm is able to recover from this missing beat and rediscover the beat structure immediately. 51
Figure 3.6: A graph of a segment of the musical signal from the rock song "Brown Eyed Girl" by Van Morrison. The tempo of this song is 76 bpm and it is significantly slower than the Celtic song displayed above in Figure 3.5. The blue lines symbolize the original signal and the red lines represent the detected beats. The intervals between detected beats are noticeably larger in this signal because of the slower tempo. 52
Figure 4.1: An example of a script file segment for the Hip-Hop animation system. The mappings between body parts in the scene and primitive movements are defined, along with the mappings between movements and musical attributes. The user can create different intervals of movements to make the animation more interesting. 59
Figure 4.2: Keywords and punctuation are extremely important to the parser when gathering information from the script file. The parser relies on both features to divide up the script details into their proper structures for later use by the animation system. 60
Figure 4.3: The pseudo code of the downHead primitive function using the Ease-In-Ease-Out function. 67
Figure 4.4: The graph above denotes a sine curve from 0 to $4\pi$. The blue line symbolizes the whole sine curve while the red line represents the segment of the curve that best describe the Jump motion. The four changes of direction in the red curve are obvious and they are used to portray the preparation, jumping, landing and post-landing motions of the Jump primitive. 68
Figure 4.5: Results from the "Jump" primitive movement. The character bends his knees to prepare for takeover, jumps into the air, and bends his knees to brace for impact upon landing. This motion follows the sine curve shown in Figure 4.4. 70
Figure 4.6: The pseudo code of the Jump primitive movement based on a sine equation. 71

Figure 4.7: An example of an unappealing movement combination where the right foot and arm are simply lifting and lowering on each beat. ..... 73
Figure 5.1: An example of a master script file using two characters. The names of the objects in the scene corresponding to the system's main body parts are specified under the CHARACTER headings. The mappings of each character to a secondary script file and each user designed routine name (SHUFFLECLICK and DUALCUT) to its corresponding text file are defined under the MAPPING heading. ..... 79
Figure 5.2: A secondary script file allows the user to design her motion sequence by specifying primitive movements, built-in routines and user designed routines in the order she wants them performed in the animation. A user can utilize loops, parentheses and rests in order to retain maximum control over the timing of the animation. ..... 82
Figure 5.3: An example of synchronizing a dance over multiple characters. Each movement in a sequence takes the same amount of time for completion as the movement directly across from it in the other character's sequence. The sequences involve movements that are different from the other character as well as the same. This is the easiest way to synchronize over the animation. ..... 84
Figure 5.4: Another example of synchronizing data over multiple characters. Unlike the figure above, the movements in this sequence are performed off-synch. The Shufflehopback routine performed by Character 1 takes 1.5 beats of time and the lines under Character 2 from Stepforward to Stamp also take 1.5 beats of time. The Jumpback primitive at the end of both sequences is performed at the same time because the movements prior to it are synchronized in time, if not in movement. ..... 85
Figure 5.5: The pseudo code for the built-in Shufflehopback routine. It uses the TapOut, TapBack, ShortHop and StampDown primitive movements. ..... 95
Figure 5.6: The pseudocode for the Stamp primitive movement, using the sine equation for smooth movement ..... 99
Figure 5.7: The pseudo code of the Cross primitive movement. The side of the body that the front foot is on (right or left) determines the direction that the movement travels. ..... 100
Figure 5.8: The diagrams above are segments from two secondary script files used to control multiple characters ..... 102
Figure 6.1: Results from the "Cut" primitive movement ..... 105
Figure 6.2: Results from the "ClickHeelsIn" (frames 1,8 and 14) and "ClickHeelsOut" (frames 29 and 40) primitive movements. ..... 106
Figure 6.3: Sequential images displaying the different positions involved in the "FrontClickJump" Celtic routine. ..... 108
Figure 6.3: These results display one way of using the Celtic system's timing aspects to combine movements. The corresponding script file is shown in the last row on the right. ..... 110
Figure 6.4: The system is easily able to accommodate multiple characters in the same scene, as demonstrated in the picture above. Sixteen girls are utilized in this particular performance. ..... 111

Figure 6.5: Results displaying how the system can use multiple characters and synchronize them all to perform the same motion at the same time. The characters in this scene are performing the "Jumpback" Celtic routine. ..... 113
Figure 6.6: Results from six characters performing unsynchronized movement. The characters are split into three groups of two, with each group performing a routine different from the other groups. ..... 115

# Chapter 1

# Introduction

Animations, whether they are in movies, television or video games, always capture the viewer's interest more if they are accompanied by music. Music has the capability to set the mood for a scene and can alter the viewer's perception of what she is seeing. The ability to tie the correct type of music in with an animation is a difficult and time-consuming process. For example, music with a dark and sinister undertone would not fit well with an animation that projects love and happiness through its movements and interactions. In the same way, a bright and cheerful song is ill suited for a sequence of events that are meant to frighten a viewer. Not only is choosing the proper type of music important, but proper synchronization of music with the events in an animation is essential when attempting to secure the attention of a viewer. An interesting animation brings with it a "wow" factor, enticing the viewer to watch and appreciate the work. This can be achieved through a good combination of interesting movements and relevant music.

One method of unifying character animation and music is through direct synchronization. This process takes an already existing sequence of motions and a piece of music and lines them up so that movements occur in time to the music. While this technique is certainly effective, it is unable to mold the animation sequence so it fits the music. The user must still choose the correct type of music to suit the animation as well as build the animation herself, either through keyframes, physics-based equations or motion capture data. This thesis proposes a method that uses musical attributes such as the beat and dynamics to build an animation that fits user specifications and is tailored to the music. The user will be able to choose any type of music she desires and create an animation that is not only automatically synchronized to the music, but also projects key elements of the music's mood as well.

Building a character animation system that is driven by music requires an efficient beat detection algorithm. Many synchronization systems use MIDI files to retrieve musical information because of the easy data extraction they provide. The main problem with MIDI files is that they are not widely accessible by everyone and it is necessary to

own specific software in order to make use of them. The system specified in this thesis uses .wav files in order to create a more user-friendly system. Musical attribute extraction is more difficult when using .wav files, but several signal processing methods exist that allow for fairly accurate beat detection. Goto built and revised a beat prediction algorithm that uses previous knowledge to determine where the next beat in a song will occur [17,18,19]. This thesis uses part of Goto's algorithm to determine beat onsets, but modifies it by combining it with a tempo detection algorithm designed by Tzanetakis [40]. This modification simplifies the original algorithm while still providing accurate beat detection.

Producing high-quality character animation has proven to be difficult for inexperienced users. Animation systems such as Autodesk's Maya and Discreet's 3D Studio Max are intimidating for a new user because of the enormous amount of features they provide. Setting up and animating a character is extremely time consuming and it generally takes practice and experience for a user to satisfactorily manipulate a human body. The system presented in this thesis provides a user-friendly method for creating a high-quality character animation where the user chooses pre-built movements to build a motion sequence. Through the use of a script file, the user can choose the order of specific movements and build a dance routine for a character, or set of characters, of her choosing. This ensures that she does not have to struggle with positioning character joints in order to achieve a specific motion. The system also gives the user the chance to experiment with different types of characters by supporting interchangeable characters. The user can change the appearance of the characters in the animation and easily use different characters in the same motion sequence. Maximum user control is provided by this system without relying on the user for the key components of the animation.

The main contributions of this thesis include:

- the development of a system that builds a new animation directly from musical attributes, rather than synchronizing an already existing animation to music.

- the implementation of a signal processing-based beat detection algorithm based on Goto's beat onset method and Tzanetakis' tempo recognition method, as well as a novel dynamics extraction algorithm.

- the development of a script file that allows for the animation of several characters and the ability to specify and build different movement routines for each character.

- the development of a mapping design that encourages the user to experiment with matching different musical attributes to different movements.

Background information and related research are addressed in Chapter 2 of this thesis. The musical analysis algorithms used for tempo recognition, beat detection and dynamics extraction are detailed in Chapter 3. The next two chapters discuss the two types of systems implemented: Hip-Hop based animation (Chapter 4) and Celtic dancing (Chapter 5). Results are presented in Chapter 6, along with a detailed explanation of the evaluation methods used to analyze the results. Future work is discussed in Chapter 7.

# Chapter 2

## Background and Related Work

Synthesizing a unique animation directly from music is a topic that has not been explored in much depth. In fact, the majority of research into this area has been done on synchronizing an already existing animation with a piece of music. Few methods discuss how to take the information retrieved from the music file and use it to directly create a new animation. Music-driven character animation involves both music analysis and movement synthesis. Music analysis is performed through beat detection and dynamics extraction. Research in beat detection has led to various options for this component of the system, but a fast and accurate method is imperative to producing a faithful interpretation of the music. Realistic character animation of a human figure is extremely important for synthesizing believable dance motion. The system discussed in this thesis relies on movements that can be combined easily and changed to reflect the mood of the music. The process of combining music and movement must be believable, with mappings that correctly match the music's impression to the impression of the movements.

In this chapter the various components that contribute to music-driven character animation are reviewed. These techniques are divided into five categories: music analysis, character motion, primitive movements, mapping techniques and synchronizing music with motion. Music analysis discusses several beat and tempo tracking algorithms that are used to retrieve information from audio data through signal processing. The following section, character motion, reviews systems that address the problem of animating a human character. Character animation is generally performed through one of three ways: motion capture-based methods, physically based animation equations and keyframed techniques. Algorithms in each of these categories are described. Music-driven character animation demonstrates that complex motions can be created using combinations of simpler primitive movements. Algorithms in other areas of computer graphics have also made this observation and their ideas will be commented on in the primitives section. Performing mapping between musical features and motion primitives

is a key component of music-driven character animation, so techniques with similar goals will be addressed in the section dedicated to mapping methods. Lastly, several synchronization techniques between music and motion are discussed in detail. Synchronization is the technique most similar to music-driven character animation because the end goal of both systems is the same: an interesting animation that moves reliably with the music. The path taken to arrive at the goal is vastly different however, and these differences will be discussed throughout this thesis.

## 2.1 Music Analysis

The major components that most listeners can distinguish when listening to a piece of music are the beat and the dynamics. The beat is a consistent pulse that sounds through the entire song and gives a sense of the tempo or speed of the music. The dynamics are the loud and soft levels that occur throughout the piece and the transitions between them. These two attributes, along with the tempo, are the musical components that we are looking to extract in our system. These attributes can be detected from either MIDI data or audio data. MIDI data can easily provide the desired information, but MIDI files are not readily available to users. We choose to use audio files rather than MIDI files for this reason.

Performing music analysis on audio files is not a trivial task. Unlike with MIDI data, it is a great deal more difficult to obtain chord information and musical note data such as pitch and tone. Audio files, however, are more accessible to all types of users and research into audio analysis has started to make good progress. The following section provides details on existing techniques for tempo recognition and beat detection, including signal processing methods employed to divide the data into more manageable components.

### 2.1.1 Signal Processing Techniques

There are several key signal processing components in every music analysis algorithm. The most popular methods for separating the signal into manageable frequency components are the Fast Fourier Transform, the Discrete Wavelet Transform and filterbanks.

The *Fourier Transform* (FT) is an extremely popular method in image and signal processing. It decomposes a signal in the time domain and outputs a representation in the frequency domain. The *continuous FT* is defined by

$$F(u) = \int_{-\infty}^{\infty} f(x)\exp[-j2\pi ux]dx \qquad (2.1)$$

where $f(x)$ is a continuous function. It extracts frequencies from the signal so that each frequency can be examined individually. If the continuous function f(x) is discretized by sampling then the continuous FT can no longer be used on the signal. A *discrete Fourier Transform* (DFT) that deals with sampled sequences is defined as

$$F(u) = \frac{1}{N}\sum_{x=0}^{N-1} f(x)\exp[-j2\pi ux / N] \qquad (2.2)$$

where the values of $u$ correspond to samples in the continuous function. In the early days of computers, the major difficulty with using the FT and DFT was that they were not suitable for implementation on a PC due to the number of complex multiplications and additions required by each equation. For an N point sequence of samples, the DFT requires $N^2$ multiplications, which is computationally expensive when N is a large number.

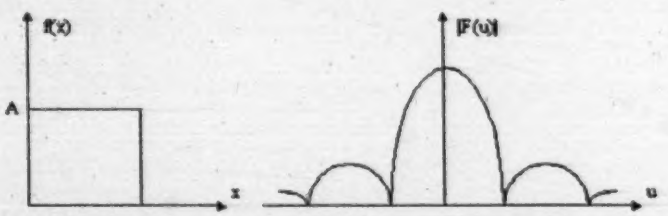



Figure 2.1: The Fourier Transform takes a time-based signal (left) and converts it to a frequency-based signal (right). These diagrams were taken from [16].

The *Fast Fourier Transform* (FFT) rectifies this problem by rearranging the DFT equation so it uses fewer multiplications, resulting in a computationally efficient equation that uses $N\log_2 N$ multiplications. An important requirement for achieving this effectiveness is that the number of samples must be a power of 2. The FFT equation is defined by

$$F(u) = \frac{1}{N}\sum_{x=0}^{N-1} f(x)W_N^{ux} \text{ where } W_n = \exp[-j2\pi / N]. \qquad (2.3)$$

This equation is easier and faster to implement on the slower computers and although today's computers are quick enough to use both the DFT and the FFT, the FFT is still the more popular method. More information on the FFT can be found in [15].

The *power spectrum* is built from the FFT and denotes a signal's power. The power spectrum represents the magnitude of the different frequencies of a signal as divided by the FFT. The most common way to calculate the power spectrum is to perform the FFT on the signal and multiply the result by its complex conjugate. However, the FFT of the autocorrelation function also results in the power spectrum. The power spectrum is used by some beat detection algorithms to determine the points at which the signal's power is increasing. This narrows down possible positions for the beat onsets.

The *Discrete Wavelet Transform* (DWT) is another algorithm that is used to separate a signal into its frequency components [15,21,30,31]. Unlike the DFT, which cannot give frequency and time information at the same time, the DWT is a time-frequency representation of a signal that preserves complete information of the signal. The signal is passed through a series of high pass filters to analyze high frequencies and low pass filters to analyze low frequencies. The signal is separated into detail coefficients using the high pass filter and approximation coefficients using the low pass filter. The approximation coefficients are often considered the most important section of a signal when performing musical analysis because the beat is generally found in the low frequency information. The DWT is extremely useful for separating out the important low frequency information from the signal so that it can be further analyzed. One pass of the DWT involves performing convolution between the signal and each filter and then downsampling by 2. Without downsampling the number of samples in each subsignal would be equal to the total number of samples in the input signal. This would result in twice as much information as needed. By resampling, complete information of the signal can be retained without storing extra samples. The frequency bandwidth for each level is also split into two, with the higher frequency bands separated into the detail coefficients and the lower frequency bands separated into the approximation coefficients.

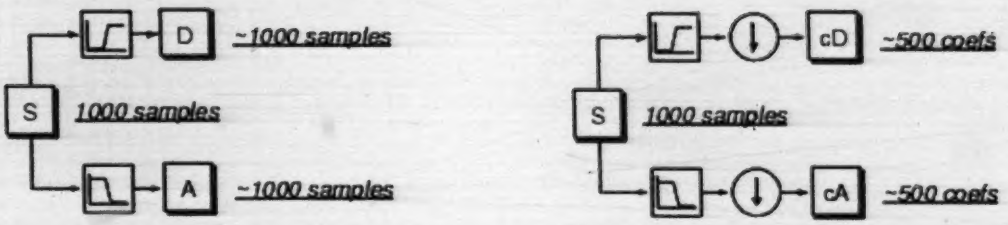



Figure 2.2: The diagram on the left displays passing a signal through a high pass filter and a low pass filter. The result is two complementary subsignals and twice as much data as found in the original signal. The diagram on the right performs the same filtering operation, but downsamples the subsignals by 2. This still provides complete information about the original signal but reduces the number of samples by half. Both diagrams were taken from [30].

The signal can be decomposed further by sending the approximation coefficients through another high pass and low pass filter set and subsampling by 2. This can continue through many levels. The detail coefficients are maintained throughout the entire process without any further processing. Figure 2.3 displays the decomposition at several levels. The same high pass and low pass filters are used throughout all the levels but the sizes change along with the subsignals. The signal is continuously decomposed into subsignals that complement each other at each level without much loss of information from the original signal. In the end, the original signal is represented by the sets of detail and approximation coefficients obtained through the hierarchical wavelet decomposition.

*Filterbanks* are also used to isolate different frequencies in a signal [15]. A filterbank is made up of a set of parallel filters that include low-pass, bandpass and high-pass filters. According to the filters used, the filterbank splits the signal into several subbands that can be analyzed separately. The number of subbands that exit the filterbank is equal to the number of filters used. Characteristically, filterbanks can include two processes: the analysis process and the synthesis process. The analysis method is used to deconstruct a signal while the synthesis method is used to reconstruct it. The analysis process filters the signal into frequency bands and then performs downsampling on each band. The synthesis process takes each frequency band as input, performs upsampling and the same filtering as used in the synthesis process, and then

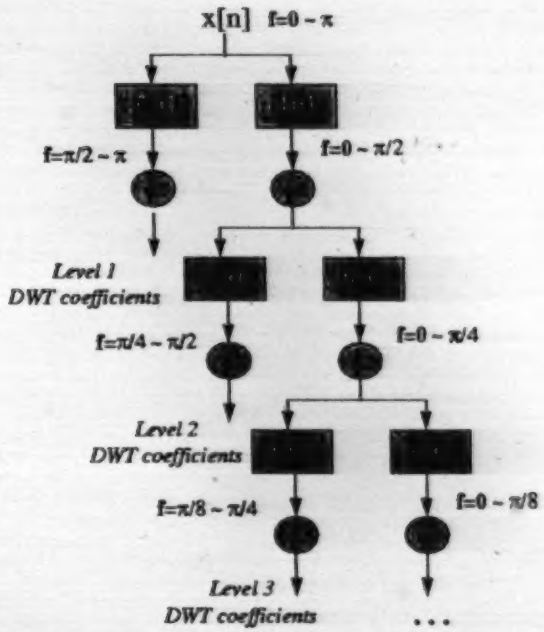



Figure 2.3: The Discrete Wavelet Transform (DWT) decomposes a signal into several frequency levels (as denoted by *f* at each level) by using highpass (g[n]) and lowpass filters (h[n]) and subsampling the results by 2. The length of the signal determines how many times the decomposition process can be performed. This diagram was taken from [31].

combines the frequency bands to generate the original signal. Filterbanks are appealing because they allow the user to choose the number of subbands to separate the signal into, as well as the types of filters used by the filterbank.

### 2.1.2 Tempo and Beat Detection

The beats of a song are the most widely recognized method of following music. It is fairly easy for even the most inexperienced listener to track beats through a piece of music. Dancing to a song is generally reliant on the beat because it provides cues for the dancer to change motions. Beat positions are the most important aspect of music analysis because they provide the structure of a piece of music. This structure is used in all dance styles and it changes from song to song. The beat structure is entirely dependent on the tempo or speed of the music. The system in this thesis uses beats to help control the

timing of the movements. The amount of time each movement receives to complete its motion is entirely dependent on the beats, so it is important that the beat detection algorithm is accurate and consistent. Beat detection for audio analysis has been studied for over a decade and many methods have been proposed. Some of these techniques are detailed in this section.

Several well-known beat detection algorithms have been proposed and refined by Goto et al. [17, 18, 19]. His techniques perform beat onset prediction in real-time using the Fast Fourier Transform (FFT) and the resulting frequency spectrum. He refines the accuracy of the onset algorithm by detecting and using chord change information and drum patterns. Beat prediction is used to determine where the next beat will occur. By using autocorrelation and cross-correlation, the algorithm looks back in time at previous onset positions and uses the calculated distance between beats (inter-beat interval) to determine the next onset position. Multiple agents are used to track different beat hypotheses across seven frequency ranges so that the system will not lose track of the beats over time due to bad predictions. Specific parameters are used to track the beats, including frequency range, beat type (strong or weak) and inter-beat interval, allowing for hypotheses to take into account different pieces of information. More detailed information about Goto's beat detection algorithm is given in Chapter 3.

The tempo of a piece of music can be a key component in tracking the beat across time. The tempo implies the speed of the beat occurrences and is used by many algorithms in order to make beat prediction easier. Tzanetakis et al. implement a fairly reliable system that is able to analyze a piece of music and return its tempo in beats per minute [40]. They use the discrete wavelet transform (DWT) rather than the Fast Fourier Transform (FFT) to represent the audio signal in the frequency-time domain. A histogram accumulates the top candidate tempos throughout the analysis of each section of the audio signal. A steady beat is necessary for this method to work because it does not track tempo changes over time. The algorithm is discussed more in Chapter 3.

Unlike Goto, most algorithms use tempo detection to reinforce the beat onset detection algorithm. Methods such as Scheirer's [33] and Dixon's [10] use the tempo in conjunction with their beat prediction and are able to track tempo changes. Scheirer uses a frequency filterbank to separate the signal into six frequency bands and a filterbank of

comb filter resonators is employed for each band to determine the strongest signal period. Each resonator has a delay time T, which will respond strongest to a signal with period T. The delays vary across frequency bands and cover the range of possible beat frequencies. The results of all the comb filters are summed across all frequency bands and the delay time of the resonator with the highest value is taken as the tempo T. Using comb filter resonators allows the algorithm to track tempo changes because when the tempo changes, the strength of the resonator of the old tempo will decrease and the resonator that corresponds to the new tempo will grow stronger.

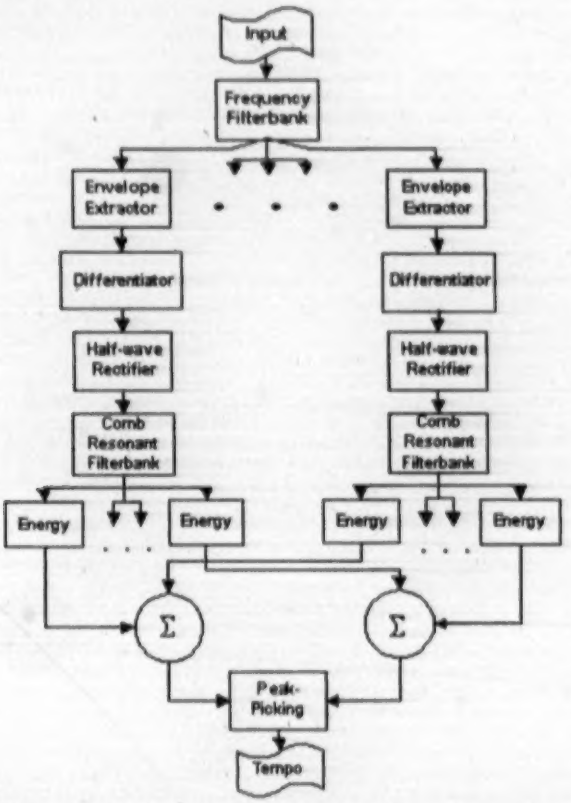



Figure 2.4: Scheirer's beat detection system tracks a song's tempo using a frequency filterbank and comb filter resonators. The frequency filterbank splits the signal into several frequency bands while the comb filters examine each band and search for a tempo that corresponds to the resonator's delay time. The diagram above was adopted from the original work.

The next beat position is predicted using each comb filter's vector of delays. The vector corresponds to the resonator's next *n* samples of output, which are its detected

tempos over the next portion of the signal. The vectors are summed up across frequency bands and the peak value corresponds to the next beat position. Many beat detection algorithms need the music to contain drums, as the low frequency sound makes beat tracking easier. Scheirer's method does not use this assumption and therefore should work with more types of music.

Tempo tracking is also performed by Dixon and used for multiple agent beat prediction [10]. Like most beat detection algorithms, the signal is divided into several frequency bands. Note onsets are detected and used to calculate the inter-onset interval (IOI). The IOIs are clustered according to note structure: half note, quarter note, and eighth note, with larger IOIs corresponding to half notes and smaller IOIs corresponding to eighth notes. The 8 seconds of music previous to the current window can be used to update tempo clusters by grouping IOIs of similar values from that period in time. The average IOI is noted as the tempo of the cluster and clusters with similar tempos are grouped together. The tempo representing the resulting cluster is updated and the 10 best tempo estimates are stored. This allows for tempo tracking and updating over time, which will result in a more accurate beat detection algorithm. Each agent is then given a tempo hypothesis corresponding to one of the clusters, as well as an IOI value from within the cluster. The signal is explored a section at a time using *tolerance windows*. Each window is separated into an inner window and an outer window. Note onsets that lie within the inner window are marked as beats, while those that fall within the outer window are stored as beat possibilities. A new agent is created for each of these possibilities, where the original agent assumes the candidate is a beat while the new agent assumes the candidate is not a beat. The creation of new agents allows for all possibilities to be considered, but also adds to the overhead of the system. Agents evaluate themselves and the beat sequence of the agent with the highest score becomes the final result of beat positions within the musical signal.

Music features other than beats and tempo have been studied and identified using audio signal processing. Like Goto, Uhle and Herre consider the structure of music, except they look for pulse levels at the note, beat and bar levels rather than strong and weak beats [41]. They identify note onsets and use the interval between notes, called the *tatum*, to detect tempo candidates, and finally the time signature. Uhle and Herre use the

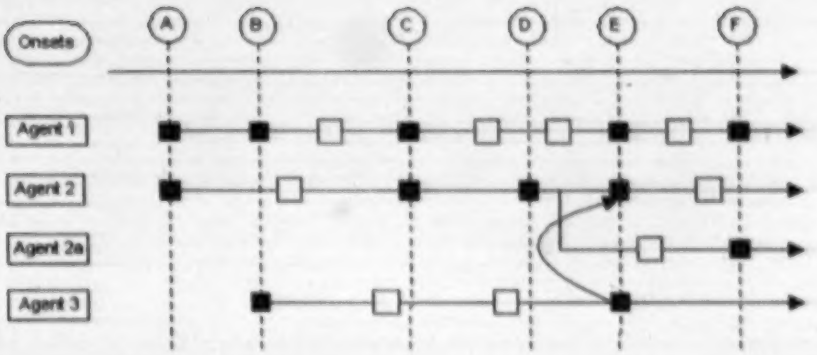



Figure 2.5: Using multiple agents to track beat hypotheses is a popular method in beat detection algorithms. In Dixon's algorithm, values A to F across the top represent the beat onsets, with the solid squares representing predicted beat times that occur on an onset and the hollow squares representing predicted beat times that don't correspond to onsets. Squares which occur close together (Agent1) correspond to a faster tempo than circles that occur further apart (Agent2). The diagram above was adopted from Dixon's original work.

assumption that the ratios between tatum periods, beat periods and bar periods are nearly integer values. This assumption simplifies the algorithm by narrowing down the candidates for the tempo and time signature. In a way similar to many other beat detection algorithms, they use a running window to split the audio signal into frequency bands. The envelope of the band is extracted and note onsets are detected using high pass filtering and half-wave rectification. Peak-picking of inter-onset-intervals (IOIs) from a histogram is used to choose the tatum period. Once the tatum period has been discovered, autocorrelation is performed on the envelope and the resulting peaks that are integer multiples of the tatum period are extracted. These peaks represent tempo candidates. The tempo candidates are subsequently used to determine bar length candidates by choosing the peaks in the autocorrelation result that are integer multiples of the tempo candidates. This algorithm is able to extract more of the musical structure than most other methods; however, it is extremely dependent on the accuracy of the note detection. Unfortunately, it is also not able to determine the beat positions of the song so it is best used for tempo detection.

Jensen and Anderson also use note onsets to help them compute the inter-beat-interval and detect the positions of the beats [22, 23]. The note onsets are detected using the high frequency content (HFC) of the audio signal. They determine that this audio

feature provides them with the best note onset results by performing several tests and computing error measures for comparison with other audio features, such as amplitude and spectral irregularity. Using the detected onsets, they generate a beat induction histogram to calculate the current beat interval. The histogram is updated for each new note onset with a Gaussian curve at the interval values corresponding to the distance between the new onset and the previous one. The maximum peak of the histogram is considered the current beat interval. The beat detection algorithm takes each note onset and calculates the distance to a previously detected beat position. If the distance corresponds to the current beat interval, then all the note onsets within that interval are compared. Using the information from the HFC analysis, the height of the onset peaks are measured against one another. If the current note onset peak is the strongest in the interval then it is chosen as the next beat and the algorithm proceeds to the next interval.

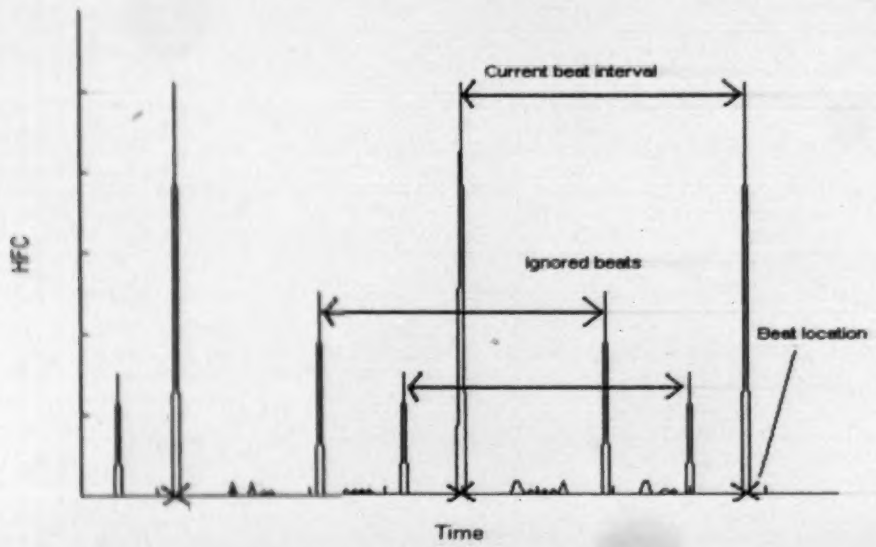



Figure 2.6: Using the previous beat and the estimated current beat interval, Jensen and Anderson's algorithm determines if the current note onset (the peak pointed to by the Beat location arrow) is the next beat in the audio signal. This diagram was adopted from Jensen and Anderson's original work.

A fairly unique method for detecting beats in an audio signal involves recurrent timing networks [20]. These networks allow a signal to be compared with itself at different points in the past. They can be used to detect patterns and periodicities and then

proceed to use that information to make future predictions. The delay loops in a network are similar to memory in that they retain the temporal structure of patterns for later comparison. Whenever the pulses entering the network as input match the pulses arriving through the loop (patterns previously detected and stored), the magnitude of the pulse entering the loop increases. This results in recurring patterns being built up in the loop, which is extremely useful with respect to detecting recurring beats. Harper and Jernigan first process the input audio signal and send a resulting pulse signal to the system. Each pulse in the signal corresponds to the position at which a sound onset was detected. Details on detecting the sound onsets were omitted from the paper. Computing distances between the current onset pulse and previous ones forms a histogram where the intervals that have the highest peaks become tempo hypotheses. Similar to the agent-based systems mentioned previously, this algorithm uses a group of detection nodes, each of which corresponds to a beat period hypothesis. Nodes evaluate their success individually and the node with the highest final score is chosen as the beat output. Each node contains a recurrent timing network where the number of delay loops corresponds to the node's beat period hypothesis. The loop in the network with the highest activation level corresponds to the next beat position. An output signal of pulses is built, where each pulse denotes the position of a beat as detected by the network. Although the concept behind this method is unique, it suffers from a problem that occurs with most other existing beat detection methods. It has problems detecting beats in jazz and classical songs, which are traditionally songs without a strong steady beat, and so its range of types of music is limited.

## 2.2 Character Motion

Realistically portraying human motion is a difficult task that has been studied through various views. Body joints are positioned and rotated from one position and orientation to the next in order to create motion over time. There are three main categories of character animation: keyframing, motion capture data and physics-based equations. In most character animation cases there exists a trade-off between complexity and realism. Physics-based equations provide physically realistic animations, but they are complicated and difficult to implement. Keyframing techniques are fast and efficient, but their

realism is based on the animator's ability to reproduce movement. Motion capture is fast to use and accurate, but it can be extremely difficult to obtain the data needed as the quality of the motion is based on the experience of the actor. Algorithms implemented within each of these categories are discussed in this section. Choosing a particular method for character animation depends on the purpose of the system, the level of realism and detail required and the speed necessary for making the system achieve its goals. All three types of techniques were considered for the implementation of our system, but keyframing was chosen because it best fits the Celtic style of dance and the goals of the animation system.

### 2.2.1 Keyframed Motion

Keyframed animation is the earliest form of computer animation. The animator sets the position and orientation of the character's body parts or joints at specific frames and the system interpolates between the frames to create an animation. It is a less realistic form of animation than physics-based animation or motion-capture data but it is generally faster and easier, especially when using an animation system such as Autodesk's Maya or Discreet's 3D Studio Max. Keyframed motion is also not subject to the same constraints as motion-capture data and physics-based animation. Unlike systems that use motion-capture data, the movements can be altered from the original and the animator is free to experiment with a huge range of motion rather than being constrained to the sequences in the motion database. Movement that looks good is not always physically correct and keyframed motion can take advantage of its lack of physical constraints to create visually appealing animations.

Thorne et al.'s Motion Doodles approach is a novel way of sketching character motion using keyframed animation and mapping [38]. The character is supplied by the user and is animated according to system specifications. The user creates the animation by using a sequence of lines, loops and arcs called *gestures*. The sketched motion is then mapped to a set of motions that will make up the desired animation. Each gesture is mapped to a specific motion where the height, start and end points and time taken to draw the gesture all make subtle changes to the movement. The gestures give the user complete control over the generation of the animation without the concern of how to

design individual movements. A parser is included in the system to split up the input gesture sequence into recognizable individual gestures. The final animation is synthesized by using keyframes and a Catmull-Rom interpolant. The root or center-of-mass of the character is placed halfway between the feet and uses parabolic curves for placement during movements that are airborne, such as jumps and flips. Inverse kinematics is used for motions where the character comes into contact with the ground. Like the system detailed in this thesis, Thorne's work encourages experimentation by the user.

Spacetime optimization is another method that can be used in combination with keyframing to synthesize new movement. Liu and Popovic built a system for rapid prototyping of realistic character motion from simple animations by using a small set of specific keyframes and constraints [29]. The input consists of a character with joints and an animation that contains joint angles at each frame. Environmental constraints, such as feet staying on the ground, are automatically extracted from the input data and physical constraints for the movement, such as gravity and momentum, are generated by the system. Momentum is used to ensure realistic motion occurs between the user-defined keyframes. The unconstrained, or in-flight, movements are separated from the constrained, or on-the-ground, movements and transition poses are used to connect constrained and unconstrained sections. The user can choose these poses or ask the system to suggest some. The spacetime objective function is built for realistic movement based on mass displacement, degree of freedom (DOF) deviation, and static balance. Mass displacement ensures natural joint movements by determining the mass displacement over the entire character. The variation between DOFs is minimized to ensure smooth movement between frames, while the static balance is important in realistic looking movements where the character is standing still. The full spacetime optimization formulation minimizes the objective function while satisfying environment, transition pose and momentum constraints.

### 2.2.2 Motion-Capture Data

Motion capture data involves capturing detailed movements performed by a real performer and building a database of the movement sequences that can be used in topics

such as character animation for reconstructing the motion with computerized characters. Motion capture systems are expensive to use and it can take a lot of time and effort to retrieve the precise movements requested by the animator. The quality of the data is only as good as the actor performing the movements and it can be difficult to modify the existing data. Despite these drawbacks, motion capture data is widely popular for character animation due to its realism and the ability for animators to reuse the data. Once large databases of various types of movements are created, motion capture data will become even more useful to character animation systems. Synthesizing new animations based on existing motion capture data can be performed in many ways and some of these methods are discussed in this section.

Splitting up motion capture sequences and using the pieces to build new sequences requires transition control. Some movements cannot possibly occur in sequential order due to extreme differences in positioning or the type of movement. Some approaches address this problem by using directed graphs to determine connections between movements. A directed graph can be fashioned by representing the vertices as individual motion sequences and the edges as transitions between nodes [2]. An edge will exist between two sequences if the last frame of the first vertex is sufficiently similar to the first frame of the second vertex to allow a transition between the movements. The graph edges are given costs to encourage the system to travel on certain paths: an edge with a smooth transition between two frames is given a low cost, while an edge with a discontinuous motion is given a high cost. In this case, the system only uses the clip sequences that it has in order to create a new motion.

Another example of a directed graph method takes the opposite approach to building the graph [25]. The edges of the graph represent the sequences of motion clips, while the vertices denote transition points. A vertex exists between two clips if they can be connected smoothly through blending techniques. In this method, the graph is not limited to the motion clips found in the motion capture data. Kovar et al. implement an algorithm that creates transition clips that can connect two segments of motion. These clips can be placed in the graph between two vertices that would normally be disconnected due to a lack of similarity in their data. This allows for a wider range of

graph paths and resulting animations. Neither motion graph algorithm makes changes to the actual frame sequences themselves, only the order in which they are used.

A similar approach to the methods detailed above include Li et al.'s *transition matrix*, whose format is that of a weighted, directed graph [28]. The motion capture data is split into *textons*, or primitive movements, where each texton is modeled by a linear dynamic system (LDS). The authors implement an algorithm to learn the motion textons from the motion capture data, as well as their relationships to each other. The directed graph is built such that each vertex is a texton and an edge's weight corresponds to the probability of transitioning between two textons. The user chooses the starting and ending textons of the animation and the system finds a sequence that passes through them by traversing the graph. This system can synthesize an animation in real-time, but learning the texture of the motion (the individual textons and their relationships) takes much longer. By dividing the original data into small primitive motions, the transition matrix will have many options for creating a path and the resulting animation.

A technique for synthesizing new animation from motion capture data that differs from directed graphs is optimization [32]. The purpose of optimization is to find a motion that minimizes an objective function and best satisfies what the user wants while still providing physically valid movements. The objective function involves three components: minimizing torques, ensuring joint angle trajectory smoothness and ensuring the resulting high-dimensional motion has angles and poses similar to those used in the corresponding low-dimensional motion. In this method, the user specifies an initial sketch of the motion through interpolation and a set of constraints, such as the starting and ending poses for the animation. Optimization is difficult when a character has a large number of degrees of freedom (DOF), so Safonova et al. solve this problem by reducing the dimensionality of the original motion. They believe that five to ten degrees of freedom can represent many human motions, rather than the sixty that are used by complex and high quality characters. Finding motions using optimization for sixty DOF is a difficult and time-consuming problem, so by reducing the DOF the complexity of the algorithm is also reduced. The user chooses movements from a motion capture database that have similar behaviour to the movements she wants performed in high-dimensional

space. The system then synthesizes high-dimensional motion that is similar to the low-dimensional movement sequence.

### 2.2.3 Physical-Based Motion

Physics-based animation is an extremely realistic form of animating a character. Motion is constrained by physics laws, joint torques and external forces such as the ground and gravity. The purpose of this type of character animation is to model the motion as closely to real life motion as possible. Although extremely realistic, physics-based motion is time-consuming to implement and few character systems exist that contain a large collection of movements.

Simple physics-based movements can be performed on a number of different character types, such as lamps, cats and bipedals [26]. User-interaction is key in this system, as the mouse and keyboard are used to control movements. Keystrokes correspond to different movement sequences set up in the system, while the mouse is able to control joint angles for body parts such as the hip and knees in a walking animation. Laslzo et al. use proportional-derivative (PD) controllers to compute the joint torques while taking into account dampening and stiffness parameters.

$$\tau = k_p(\theta_d - \theta) - k_d\dot{\theta} \tag{2.4}$$

$\theta_d$ represents the desired angle, $\theta$ represents the current angle and $\dot{\theta}$ represents the angular velocity. The dampening and stiffness parameters are $k_p$ and $k_d$. Their state machines combine similar movements into a single action group where the user can choose a movement, such as taking the next step, and the state machine will choose which movement in that group will be used. In most cases, the choice of the next movement in physics-based animation is a direct result of past movements, so the authors provide a *checkpoint* that allows the user to save the animation up to a point in time and return to the end of the saved work to rework the next section of the animation if it does not suit her standards. Laslzo uses fairly simple movements and gives the user the ability to control the entire physics-based animation through mouse movement and keystrokes.

One system that has incorporated a number of physics-based movements is that of Faloutsos et al. [12,13]. They create physics-based controllers that are called based on

their suitability to perform the required action. Controllers can be integrated for simple movements, such as balancing and stepping, or they can be implemented for complex movements, such as walking and rolling over. Each controller comes with pre-conditions and post-conditions that must be met in order for the controller to be chosen by the system and successfully complete its objective. Pre-conditions involve the initial state of the figure, environmental parameters such as whether the feet are in contact with the ground, the character's balance, and the target state. Post-conditions include similar parameters to the pre-conditions; except these parameters contain values that should be met by the time the controller is finished.

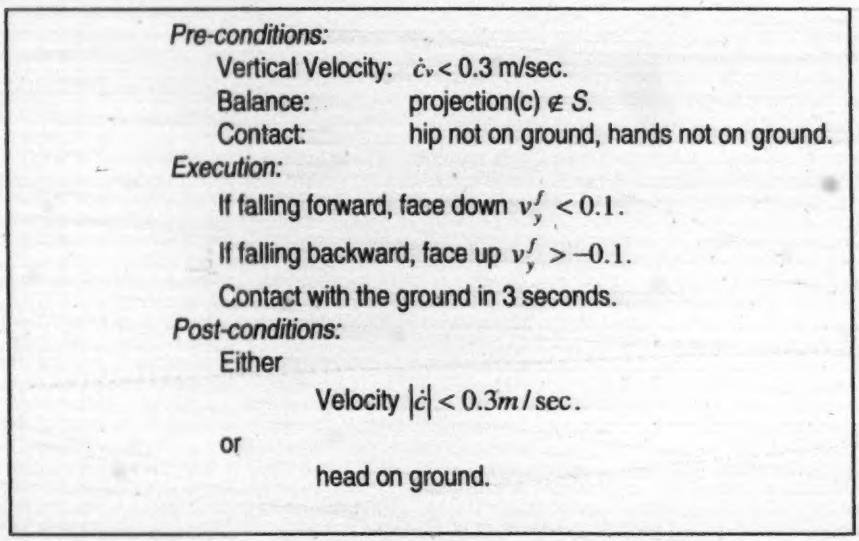



Figure 2.7: Pre-conditions, expected performance and post-conditions for the physics-based Fall controller designed by Faloutsos. The controller takes into account velocity, balance and environmental parameters such as ground contact. This controller diagram was redone based on Faloutsos' Fall controller.

Each controller rates its expected performance for the next task and this evaluation is used by the system to decide which controller will be chosen next. Transitions between controllers occur naturally in three instances: when a controller has finished, when the user intervenes or when the controller detects that it has failed. The controller that best suits the next piece of motion is then called by the system and used to continue the animation.

## 2.3 Synchronizing Music with Motion

The purpose of most synchronization methods is to take an already existing animation and synchronize it so that movement changes line up with beats in the given piece of music. The result should be well coordinated so that it appears as though the animation was originally built off of the music, rather than changed to match the music. In most cases it is the animation data that is changed to suit the music, rather than the music to suit the animation, so these techniques will be the ones mentioned in this section because they are most similar to the system discussed in this thesis.

### 2.3.1 Synthesizing New Motion

Motion capture data is the most popular form of character animation because it is realistic, easily used once retrieved, and can be manipulated through editing and blending techniques, such as time-warping. The motion data can be split into smaller sequences and rearranged to form new animations. An approach proposed by Kim et al. [24] uses a *movement transition graph* to synthesize new motion sequences. In order to synchronize the animation with the musical data, they look for the rhythm in the motion and match it to the rhythm in the music. Moments where joints perform an obvious change in direction are termed *motion beats* and these beats are used to split up the original motion data into smaller basic movements. A clustering technique is used to group similar motion beats and the best representative of each group becomes a node in the movement transition graph. The edges in the graph represent the transitions between clustered groups and are based on how smoothly the transition between movements occurs, as well as whether such a transition fits the rules of the corresponding dance. The transition graph is traversed and movements at each node are blended together until the end of the dance is reached, resulting in a new formation of the original data. The dance is then synchronized with the input MIDI data by timewarping the animation so the motion beats and music beats are aligned. A major problem with this synchronization technique is that the tempo of the motions beats must match the tempo of the song. This restricts the types and speeds of songs that could be used as input and does not allow the user much freedom in testing dances with different styles of music. Transition graphs are labeled based on the type of dance and the rhythmic pattern to which its movements correspond

(eg: Waltz, Salsa). This setup does not permit experimentation with putting together steps from different dances to produce a new style of dance.

Another method that uses motion capture data and a transition graph to organize the sequences of motion data is found in the work of Alankus et al [1]. The motion data is analyzed for frames that are similar to each other. Frames are similar only if an arbitrary translation on the XZ plane and an arbitrary rotation on the Y axis exist such that the transformed points in one frame $f_i$ are closer to the non-transformed points in another frame $f_j$ than a threshold $\varepsilon$. The transformation allows them to compare character poses that are similar except for the position and rotation of the character. This leads to the assumption that if two frames $f_i$ and $f_j$ are similar, then frame $f_i$ can easily make a transition to $f_{j+1}$. The transition graph is built to represent movements between *dance figures*. A dance figure is defined as a sequence of frames $f_i...f_j$ from the motion capture data where there are frames in the motion capture data that are similar to the beginning frame $f_i$ and the ending frame $f_j$ of the sequence. This is necessary because the system needs to be able to make a transition from this figure to another one in order to create an animation. The authors only want frame transitions to occur between dance figures, so the definition of a dance figure is further constrained to a movement sequence in which there are no motion capture frames that are similar to any frames between the starting and ending frames of a dance figure sequence. In a similar process to the extraction of *motion beats* in Kim et al's method [24], *dance moves* are identified based on significant changes in the movement of a body part. The dance moves are synchronized to the music data by traversing the transition graph and finding the sequence of dance figures that best fits the music. A dance move is chosen if its timing can be changed by increasing or decreasing the speed of the frame to fit the beat position. Both a greedy algorithm and a genetic algorithm are used individually to improve the results.

The methods mentioned above synchronize a new animation without much regard to the input music. The musical rhythm is used to speed up and slow down movements to provide a well-synchronized animation, but the resulting motion sequence is chosen based only on the beat structure of the music and not its mood or expression at a point in time. Shiratori et al. [35] propose a new approach to music-motion synchronization that uses musical features to choose motion segments that best match segments of the music.

Shiratori's group believes that in real life the rhythm and intensity of dance movement is synchronized to that of the music. They define the intensity of a musical piece as the excitement level. For example, a quiet ballad has a low intensity level, while a hard rock song has a high intensity level. The rhythm and intensity components are detected in both the music and the motion data and used for synchronization. Once the motion capture data and the music have been divided into segments, the rhythmic similarity between a motion segment and a music segment is determined. This results in candidate motion segments for each music segment. The list of candidate motion segments is put through connectivity analysis to determine if the pose and movement from one segment to the next is similar. The outcome of the analysis is a set of candidate segments that fit the music's rhythm and have natural transitions between each other. Shiratori uses a Bhattacharyya coefficient to evaluate the intensity similarity between the music and motion components. The final result is a motion sequence that fits the music's rhythm and intensity and has natural looking transitions between segments. This sequence can be combined with the music for a newly synthesized animation that better suits the music. Unlike this thesis work, their algorithm does not run in real-time and like the methods mentioned previously, the algorithm only works well if the rhythm and intensity features of the motion data are similar to the input music. If the music is not similar in timing or in excitement level to the motion data provided, then realistic synchronization will not occur.

### 2.3.2 Direct Motion Editing

Creating new animations for synchronization purposes does not always involve rearranging motion capture data. Motion curves that the character will follow can be altered using motion-editing techniques, resulting in a curve that changes as the music does [6]. Cardle et al. implemented a system that contains several different motion editing techniques, including filter banks, additive motion techniques, motion warping and time warping. Filter banks divide the motion signal into components that can individually be changed and put back together. Additive motion techniques blend motions together, while motion warping blends a displacement map with the motion curve. Time warping increases or decreases the speed of a sequence of frames so it can

be better synchronized with the music. The user chooses the music feature to motion editing mappings, where a specific music feature will cause its corresponding motion editing technique to be performed on the motion curve. For example, motion warping can be mapped to the musical beat by adding a point to the displacement map for each beat, resulting in a jump in the signal at each displacement point. This mapping method allows for interesting changes to the motion data and gives the user the ability to experiment. Cardle et al. prefer to use keyframed animations rather than motion capture data, although they claim that their system allows for both, because more motion editing techniques can be performed on keyframed animations. This method is a much simpler way of synchronizing music with animation than those of Kim et al. and Alankus et al. because it does not make large changes to the motion data, such as rearranging movements. The mapping method of Cardle et al. gives more user control than the previous two and does not place restrictions on the movements or music given by the user.

Goto mentions beat-driven real-time computer graphics in the form of his dancer Cindy, which is synchronized to a dance sequence with music using his beat-tracking algorithm [17]. The system contains pre-defined dance sequences and the user selects one to synchronize with music. The timing of the sequence is automatically changed to reflect the timing of the beats as detected in the music. This system has many similarities to the one discussed in this thesis, but Goto does not provide much information on the details of the system or any results, so it is difficult to compare his system with that in this thesis.

### 2.3.3 Detecting Motion Features

Changing the motion data in order to synchronize it with the music can cause drastic changes to the timing and appearance of the movements. To address this problem, Lee and Lee propose a method that changes the timing of both the motion capture data and the music [27]. Their feature mapping method is an extension of Cardle's approach, where both methods allow the user to select the music and motion features. Motion features included in this system differ from Cardle's system in that they are movements in the animation rather than editing techniques. Lee and Lee use features such as

footsteps, arm-swinging motion, and occurrences when motion has stopped in the animation. The footsteps are detected by using the vertical position of a foot and finding the local minimum points in the movement. This is achieved by detecting the zero crossing points of the first derivative where the second derivative is greater than zero. Arm-swinging motion is found by looking for the local maximum points of the arm movement, or the points where the arm is at the end of a swing motion (either in front or behind the body). Using the Kinematic Centroid Segmentation technique, Lee and Lee find the motion curve $a_2(t)$ of the arm from the following equation:

$$a_2(t) = (C(t) - B(t))^2 \qquad (2.5)$$

B(t) is the position of the arm's shoulder at time t and C(t) is the average positions of the wrist, elbow and shoulder at time t. Dynamic programming is used to pair the music and motion features that are closest in distance. Synchronization between the music and the motion occurs by time-scaling the music features to match their corresponding motion feature positions. Musical feature points are discarded if they change the music's tempo too much, but are later synchronized by time-warping the motion.

Shiratori et al. propose a similar motion feature detection method that follows the position and speed of the arms, legs, and centre of mass (CM) [34]. Shiratori believes that dancing is composed of many primitive movements. Unlike Lee and Lee, Shiratori et al. uses the beat features from the music to help extract primitive movements in the motion. Shiratori's method assumes that for dancing, a keyframe occurs at the point where the dancer momentarily stops dancing, which is usually at a beat. Once these keyframes have been detected for the feet, arms and CM, the keyframe candidates where the entire body has stopped moving are chosen as the motion features. The positions of the motion features are then used to segment the motion sequence into primitive movements and could be used to line up the primitive movements with the music beats.

### 2.3.4 Keyframed Techniques

Not all interesting character animation techniques involve motion capture data, as shown by Taylor et al.'s work [37]. By using the ANIMUS framework for a virtual character, they map musical features extracted from MIDI data to the character's behaviour. The user can input the music into the system through a piano or by singing into a microphone.

The non-human character is not controlled by motion capture data, but is still able to convey the emotional significance of the music through its movements. The character responds to the music according to the mapping functions designed by the system designer. The musical features used by the system include pitch, amplitude, chord information and vocal timbre. As these features change throughout the musical data, the character's movements will change as well. For example, if the character hears the sound of a particular pitch, she will look around in the environment for the origin of the sound. The user is able to interact with the character through the music, making it an interesting addition to musical performances; however, the user cannot change the mappings or the character itself without help from the system designer.

## 2.4 Labanotation

Labanotation is a system for analyzing and recording human movement [5]. Its movement notation is used in the dance community to record and analyze dance movements and sequences. It is not specific to a particular dance style, but allows choreographers to cover all ranges of movement. Similar to musicians using musical scores to create songs, choreographers use labanotation to plan combinations of motions and dances. Each symbol in labanotation consists of four pieces of information. The shape specifies the direction of the movement; while the shading indicates whether the movement is performed at a high level, middle level, or low level. The placement of the symbol on the dance staff determines which body part is performing the movement, and finally, the size or length of the symbol indicates how long the movement is performed for.

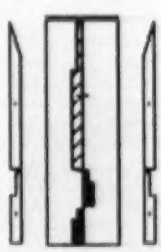


Figure 2.8: Example of Labanotation from the Dance Notation Bureau website [9].

The notation is fairly complicated and can take a long time to write out and as a result, only a few people in the dance community use it faithfully. Several animation systems have entered the market that allow users to translate labanotation to a 3D animation (LabanDancer [42]) or create dance scores by choreographing dances completely with 3D characters (DanceForms [8]). Labanotation is outside the scope of this thesis and will not be discussed further.

## 2.5 Motion Primitives

It is the belief of the author of this thesis, along with others [14,42], that complex motion can be simplified into a combination of basic movements called *primitives*. Dancing is a real-world example that supports this theory. Long dance sequences can be split into routines that consist of separate dance moves. The individual dance moves are the primitives that are combined together to create dance routines and performances. Identifying primitives is a difficult task, one that is performed to some extent by motion segmentation as discussed above [2,24,26,27]. The majority of these methods segment motions by identifying velocity changes between frames.

Fod et al. also implemented an algorithm for automatically detecting and segmenting primitives from movement data [14]. Their system uses only arm movements and motion data built from the joint angles of the arm to generate primitives to describe the arm movements. Using an imitation model to animate the character, they build the animation in several layers. The first layer, the perception layer, acquires the movement data and selects meaningful data from the motion stream, such as significantly moving features and prominent kinematic substructures. The encoding layer classifies movements into primitives, refines the primitives, and encodes the movements using the primitives. This layer outputs two components: the segment sequences that describe when a primitive occurs and the set of constraints for creating primitive controllers. The last layer, the action layer, performs the imitation by executing the segment list. Segmentation of the motion data into primitives is done in two ways by the authors. The second method is a more effective and accurate algorithm, where segmentation is performed based on thresholding the angular velocity of a motion primitive. PD controllers are used to execute the primitives by computing the necessary torque for the arm joints.

A method that exists outside character animation for characterizing primitive movements is built from the kinematic theory and its deltalognormal model ΔΛ [43]. The kinematic theory can accurately describe kinematic relationships as well as complex movements. It also allows an accurate depiction of human movement. The ΔΛ model is expressed in terms of speed and is used by Woch and Plamondon to express five different types of velocity profiles. Rather than segmenting motions entirely based on a single velocity peak (or the lack of velocity at a specific point in time) as is done by most primitive segmentation algorithms, this method can discover primitives with up to three velocity peaks and up to two direction reversals. This allows for slightly more complex primitives that would otherwise be described by several smaller primitives, which in turn encourages motion sequence representations that are based on fewer primitives. This method supports our belief that primitive movements can be more complicated than simply changing direction or velocity. For example, the Hop primitive in our system includes three directional changes and would not be found properly by most segmentation methods. Woch and Plamondon's method presents a new way to describe more complicated primitives that can still be used to build complex motion.

## 2.6 Mapping

Mappings between features are extremely useful when using input data to drive an animation. In the system discussed in this thesis, the musical attributes are mapped to character movements. Similar mapping methods are used in facial animation to map vocal attributes to facial movement, as well as in motion retargeting where the movements of a performer are mapped to the movements of an animated character. Using a concept similar to ours, many voice-driven facial animation systems analyze the sound data and use certain features to drive the facial expression of the character. These approaches give merit to our method and provide a good basis for comparison with other data-driven animation systems. Motion retargeting provides another type of mapping that uses important character features to create novel animation results. The nature of this mapping is similar to a mapping method detailed in this thesis because both focus on correctly matching movement to the corresponding character. Both types of mapping,

input features to movement and movement to the character in the scene, can be found in music-motion synchronization.

Brand and Shan [3,4] attempt to drive facial expressions from vocal information by learning from synchronized sound and video data. One of the goals of their system is to learn mappings between vocal features and facial features for synthesis purposes. A Hidden Markov Model represents the positions and velocities of facial features obtained from Brand and Shan's vision system, with each state representing a particular expression. Training video data helps the system determine the most probable set of facial states and the vocal features are then mapped to these states. Mapping between expression and vocal information is achieved by computing the probability that a certain vocal feature can be associated with a facial state. To synthesize facial dynamics for an input vocal track the expression-voice mapping computed in the learning stage is used to find the most probable sequence of matching expressions. Each state in this resulting expression sequence is mapped to a final facial configuration in order to animate a character. Like our method, their mapping technique uses features extracted from audio data to drive the animation and create a unique sequence tailored to the input audio.

Another method of mapping in facial animation research is that of mapping between facial expressions. Rather than determining the facial expression from the audio data, Cao et al. [7] look at transitioning between two expressions within the same sentence. The system is able to automatically determine the emotional expression of speech signals using supervised machine learning techniques and classification methods. Five types of emotion expressions are used to make up the emotional spaces: neutral, sad, angry, happy, and frustrated. Cao's emotional mapping function determines a transition between two facial motions in two different emotional spaces. This is done through the use of a training set that includes sentences where motion transitions through all five emotional spaces and a Radial Basis Functions mapping function. Cao's mapping procedure allows for arbitrary sentences to transition smoothly between facial expressions, resulting in an animation that expresses emotion corresponding to the audio content.

Computer puppetry is a form of motion retargeting that takes motion capture data and determines how to properly map it to an animated character with a different size and

proportion than that of the actual performer [36]. This method is not as simple as directly mapping joint angles and end-effector positions from the performer to the animated character because the body proportions change the amount of movement necessary to reach the same goal position. Shin's system uses an importance-based approach to automatically decide what the important features of the input movement are, as well as how to use these features to recreate the input motion. The three features targeted by the algorithm are the character's root position, the joint angles and the end-effector positions. The system will choose which feature is most important to the movement and move the character in order to preserve the feature's data. Shin's method preserves enough of the original motion by mapping the input feature data to the character data, but it is also able to change the motion to best reflect the character's proportions.

# Chapter 3

# Music Analysis

Music analysis is performed by combining two different algorithms: Tzanetakis et al's tempo detection method [40] and Goto's beat tracking method [17,18,19]. Tzanetakis' method was faithfully followed in the implementation, but several changes have been made to Goto's method in order to make it work better for our purposes. The algorithms are detailed in this chapter along with our version, which includes collaboration between the two.

## 3.1 Tempo Detection

The speed of a song is an extremely important musical feature to composers and listeners alike. The speed, or tempo, dictates the positions of the beats in a song and describes how quickly the rhythm of the song will be performed. Corresponding dance movements are also affected by a song's tempo as their speeds are influenced by the beat positions. We have implemented a tempo detection algorithm from Tzanetakis that is used in our beat detection method to determine the positions of the beat in the music. Details of Tzanetakis' algorithm are presented in the next subsection. The results displayed include histograms for two different types of music: rock and Celtic. The structure of all tempo histograms is the same, so only two are shown for recognition purposes.

### 3.1.1 Algorithm Details

The tempo detection method we employ for the first step of our beat tracking method uses the Discrete Wavelet Transform (DWT) to decompose the signal into a number of octave frequency bands. The DWT is able to represent the signal's information in time and frequency space, rather than just the frequency space like the Fast Fourier Transform. A pyramidal algorithm is used with the DWT to split the signal into several frequency bands. Tzanetakis decomposes the signal into twelve levels of coefficients, but our implementation only decomposes into five levels. We use only five levels of coefficients

because experimentation found that five is the smallest number of levels with the same effectiveness as that of twelve levels.

A windowing technique is used to divide the signal into small, equal-sized windows that are analyzed one at a time by the algorithm. The results from the analysis of each window are combined into a histogram. A window size of 65536 samples is used and while Tzanetakis uses a frequency sampling rate of 22050 Hz, we use a sampling rate of 44100 Hz because it includes most audio signals. There is no noticeable difference between using Tzanetakis' sampling rate and ours. The step size for moving the window is 512 seconds or 32768 samples. The signal in each window is decomposed by the DWT, resulting in five sets of detail and approximation coefficients. We keep the approximation coefficients and discard the detail ones because the approximation coefficients hold the low frequency information, which is where the beat is generally found in a piece of music. Due to the nature of the DWT, the number of coefficients at each level is different, making it difficult to perform further analysis and combine the information together. Upsampling is used to ensure all coefficient levels have the same number of values. Each level is upsampled by a value of $2^i$, where $i$ is the level number. This cancels out the downsampling performed by the DWT. Upsampling will sometimes cause the signal at each level to be longer than the window size. The values at the end of each level are discarded until the signal is the same size as the window, resulting in five equal-sized levels of approximation coefficients. We can then proceed to perform the steps for the tempo detection algorithm.

Full wave rectification is the first and simplest step, in which the absolute value of all the samples in the signal is taken.

$$y[n] = abs(x[n]) \tag{3.1}$$

Low pass filtering of the rectified signal is performed according to a one-pole filter with an alpha value of $\alpha = 0.99$. The one-pole filter was implemented according to the definition and parameters given by Tzanetakis. This filtering operation goes through each sample and computes its filtered value by taking into account the previous sample's filtered value.

$$y[n] = (1-\alpha)x[n] - \alpha y[n-1] \tag{3.2}$$

Downsampling is then performed on the low-pass filtered signal. A downsample value of 16 is used where $k = 16$ in Equation 3.3. Downsampling is performed to reduce the computation time of the algorithm. The combination of full wave rectification, low pass filtering and downsampling results in the amplitude envelope of each band of the original signal. Extracting the envelope of a signal is a common technique in beat detection algorithms because it is a simple method for detecting the spikes in the bass sounds that can correspond to beats. It is important to note that downsampling results in a loss of high frequency information. Indeed, the Shannon-Nyquist theorem proves that a band-limited signal must be sampled at at least twice the cut-off frequency in order for the samples to accurately represent the original signal. After downsampling by 16 the highest frequency that can be represented, according to the Shannon-Nyquist Theorem, is only 8 Hz. This could become problematic if the beats reside in the high frequency information because then they would not be detected. This algorithm is designed based on the assumption that beat information occurs in low frequency data, which is true in most cases.

$$y = x[kn] \tag{3.3}$$

Mean removal or normalization is performed on each amplitude envelope before the autocorrelation stage. The mean value and standard deviation (std) of each frequency band is calculated and the following equation produces a centered signal. The resulting frequency bands are summed together into one envelope.

$$y[n] = (0.5 * (x[n] - E[x[n]] / std)) \tag{3.4}$$

An important component that is incorporated into our implementation is the padding of zeroes around the summed envelope so that it is the same size as the original window. This addition is necessary for avoiding wraparound error in the autocorrelation function. The computation of the power spectrum of the summed envelope is used for the autocorrelation step. This is achieved simply by taking the FFT of the signal, multiplying the result by its complex conjugate and performing the inverse FFT on the outcome. Autocorrelation results in a symmetric signal so we remove the second half of the result and concentrate on the data in the first half.

Our implementation for detecting the highest peaks of the autocorrelation function differs from Tzanetakis, so we discuss the details here. For each window the highest 30

peaks, $P$, of the autocorrelation function are stored and used to find the top 3 peaks, $T$. We do not simply take the highest 3 peaks of the signal because in some cases these peaks can be extremely close to each other. The number 30 is large enough that 3 distinct peaks should almost always be found in the window, but not so large that the algorithm will constantly be looking for peaks where none exist.

We want a wide range of tempos so we can be sure that we are detecting the correct one. We choose the highest peak in the list as the first of our top 3 peaks, $T_1 = P_1$. We then proceed to compare the second highest value of the 30 peaks, $P_2$, with $T_1$. If the distance between these peaks is greater than 60, then $T_2 = P_2$, otherwise $P_2$ is discarded. The threshold of 60 was chosen through experimentation with different songs and found to be a large enough distance that a peak outside the threshold would not be significantly close to another peak. We go through the autocorrelation list until 3 peaks are found that are sufficiently far apart but within the appropriate range for tempo detection, which is between 40 and 200 beats per minute (*bpm*).

The three values in T are converted from sample numbers into beats per minute by the following equation:

$$S_i = 60/(P_i \cdot (k-1)/Fs) \tag{3.5}$$

where $S_i$ is the converted tempo in bpm, $Fs$ is the frequency sampling rate of 44100 Hz, $k$ corresponds to the number used to downsample the envelope (k = 16 from Equation 3.3) and $1 <= i <= 3$ for each value in T. These three tempo values are then added to the tempo histogram that tracks the tempo values detected in each window of the signal. Having one bin for every possible bpm value between 40 and 200 arranges the histogram structure. The amplitude of each peak in T is added to the beat histogram at its corresponding tempo bin. For example, if one of the values in S is 80 bpm, then the amplitude value at its corresponding peak will be added to the histogram in bin 80. The complete analysis of the entire signal over all the windows will result in a histogram with peaks in certain tempo bins. The tempo bin with the highest amplitude is chosen as the overall tempo of the song because its periodicity was detected the most throughout the analysis.

In some cases the tempo analysis algorithm will detect the wrong tempo as the estimated tempo of the audio file. This generally occurs when one of the two most

detected tempo values is double the other one. When the audio signal is in 4/4 time, it is possible to have two types of tempos: the first corresponds to every single beat (4/4 time) and the second corresponds to every second beat (2/4 time). To account for this difficulty, we compute the difference between the amplitude of the two peaks in the beat histogram. If the difference is between an amplitude interval of 500 and 5000 units then the estimated tempo of the audio file becomes the second highest peak in the histogram rather than the first. The values of 500 and 5000 were acquired through extensive manual experimentation with different musical songs. A user with musical experience listens to each song and determines if the tempo is slow or fast. The user-estimated tempo is then compared to the system-estimated tempo. In the cases where the system chooses the wrong tempo, the system designer compares the peak values of the two tempos. In the majority of cases, the system chooses incorrectly when the difference between the two peaks is between 500 and 5000 units, so this interval is used to correct the tempo choice.

The process detailed above, from the DWT decomposition to the organization of the beat histogram, is performed for every window in the signal. The tempo of the music file is only determined after the entire signal has been analyzed.

### 3.1.2 Tempo Detection Results

The tempo detection algorithm is extremely efficient. It runs faster than real-time and is very accurate. This technique was first tested on the four songs provided on Tzanetakis' website [39]. These song types include hip-hop, rock, jazz and classical. We were able to generate close to the same results as Tzanetakis, with a difference of less than 3 bpm for the peak tempos. This disparity is most likely due to the difference in the Discrete Wavelet Transform implementations used by the respective algorithms. Our music analysis system is implemented using Matlab, including the DWT and FFT functions. Tzanetakis' system is built from his own C code, which may explain the difference in results. We also choose to use the power spectrum to compute the algorithm's autocorrelation step, while Tzanetakis uses Equation 3.6. In the majority of the cases tested, both algorithms give the same results.

$$y[k] = \frac{1}{N}\sum_{n=0}^{N-1} x[n]x[n+k] \tag{3.6}$$

This algorithm does not do well with Jazz and classical songs because the beat is not prominent in these types of music. These songs often do not have a strong beat because they rarely use drums. A weak beat is extremely difficult to track, which results in an inaccurate tempo. Another difficulty with this algorithm is its inability to track tempo changes. Although most popular songs do not incur changes in speed, some Celtic songs have a beat that varies over time due to the nature of the performance. Adjusting this tempo detection algorithm to take speed changes into account is a component of our future work.

Figure 3.1 below displays the resulting beat histogram of a Celtic song named Siamsa. The highest peak is found at 117 bpm and the second highest peak at 58 bpm. The second tempo is half the speed of the first because it detects periodicities on every second beat rather than every beat. For example, if the song is in 4/4 time, a beat will be detected on every quarter note. Beats can also be detected on the first and third quarter notes to describe a song in 2/4 time, which is half the tempo of the 4/4 time. The song is 4 minutes and 29 seconds in length and the tempo detection algorithm took 3.0156 minutes to determine the tempo.

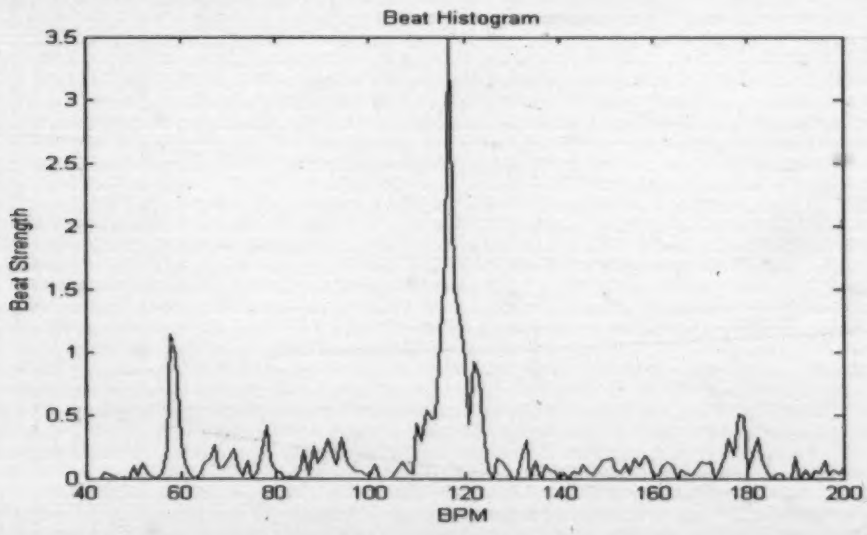



Figure 3.1: An example of a beat histogram produced by the tempo detection algorithm. The values across the bottom represent the possible tempos. For this particular Celtic song, "Siamsa," the tempo is detected to be 117 beats per minute.

One of the difficulties with the tempo detection algorithm is that occasionally the tempo with the second highest peak is the actual tempo of the song, rather than the one that corresponds to the highest peak. This can be seen in Figure 3.2 where the highest peak corresponds to a tempo of 161 bpm and the second highest peak represents a tempo of 82 bpm, which is the correct tempo of the music. The music clip is 30 seconds in length and it took the algorithm 18.656 seconds to identify the tempo. This audio file is from a song by the Beatles and is taken from the testing set of Tzanetakis that is posted on his website [39]. Both our implementation as well as Tzanetakis' identified the incorrect tempo for the song. When this problem occurs, the actual tempo is generally the second highest peak in the histogram.

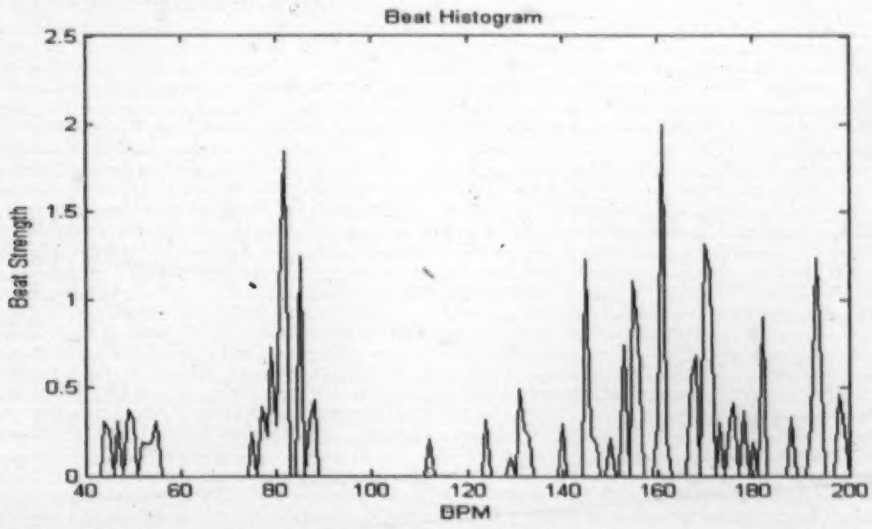



Figure 3.2: A beat histogram for a 30 second Beatles song. The tempo is identified as 161 bpm when, in actuality, the tempo is 82 bpm. The algorithm recognizes the faster beat as the tempo because it occurs more often in the signal.

The tempo of a piece of music is directly related to the speed that the beats occur in the song. It is possible for multiple tempo values to be accurate in describing a piece of music, but only one is actually correct. For example, say two listeners are tapping along to the Beatles song from Figure 3.2. Listener 1 is tapping to the correct tempo (82 bpm), while Listener 2 is tapping twice as fast (161 bpm). In terms of music, both listeners are accurately tapping to a beat. The difference between them is that Listener 1 is tapping to beats 1 and 3, while Listener 2 is tapping to beats 1,2,3 and 4. In this case, beats 1 and 3

are the strong beats, while beats 2 and 4 are the weak beats. Strong beats are generally the obvious beats in a piece of music, while the weak beats are the underlying beats that occur between the strong ones. Tapping to all 4 beats is accurate, but tapping to only the strong beats is more accurate, therefore the tempo corresponding to the strong beats is chosen as the correct tempo. It can be tough for human listeners to pick the tempo corresponding to the strong beat, which is why many algorithms have difficulties. Currently a user with musical experience chooses the correct tempo.

This approach is not the only method to have difficulties with relationships between tempos. Scheirer's tempo induction algorithm also retrieves tempos that correspond to strong and weak beat levels in a song [33]. Errors occur due to the algorithm's inability to understand the relationship between different beat levels, such as the ones that exist between a strong beat and a weak beat. Dixon's method addresses this problem by grouping together tempos that are an integer multiple or divisor of each other [10]. He uses this knowledge of relationships between beat types to create several tempo hypotheses, each of which is tracked by a different agent in order to establish the correct one. Unfortunately, the paper does not mention how successful the algorithm is in determining the right tempo. Choosing the true tempo from several accurate tempos is a problem that each tempo detection algorithm must face. The addition of extra musical knowledge into an extraction method may assist in solving this dilemma.

## 3.2 Beat Detection

The most recognizable component of a piece of music is the underlying beat structure that accompanies it. Dancers move to the beat, listeners tap a foot to it and musicians compose according to it. The beat is very dependent on the tempo of a song because the beats are positioned based on the music's speed. Fast music will include a beat with little time between positions, while the beat of a slow piece will have large interval times. Tracking a beat through an audio signal is a difficult task that has been studied in numerous forms.

Goto's algorithm [17,18,19] was chosen for our musical analysis system because it is referenced in every beat detection paper and is considered by many to be an excellent technique for performing beat detection. There exist a few papers explaining the

algorithm with different levels of detail, which makes it easier to implement and replace some of the components with our own. Unlike other beat detection algorithms, this one can be used on drumless music, which makes it more robust for possible input songs. This section outlines the algorithm used to detect the beat onsets in the music and displays results for music of different types and speeds.

### 3.2.1 Original Algorithm Details

Our original beat detection algorithm uses primarily the tempo to predict the next beat position in the signal. The tempo can easily be converted to an inter-beat-interval (IBI) that describes the distance between beats. The first beat in the signal is detected using a simple version of Goto's algorithm and each subsequent beat position is predicted by adding the inter-beat-interval to the previous beat position. More details on Goto's algorithm are given in the next subsection.

This algorithm divides the signal into large sections where each section contains approximately 5 beats (as determined by the tempo). This is done to reduce the amount of data that the algorithm will examine at one time.

$$windowSize = IBI * 5 \tag{3.7}$$

The window size is a multiple of 5 beats because this value gave the most accurate results during testing. Window sizes ranging from 2 beats to the entire signal were tested and sections consisting of 5 beats worked best.

Goto's beat detection technique is used to determine the position of the first beat in each section. The highest of the first 10 peaks in the section is chosen as the first beat position. The next beat position in the section is predicted by adding the IBI to the first detected position. The third beat position is found by adding the IBI to the previously detected position, and this continues until all 5 beats in the section are determined.

$$b_i = b_{i-1} + IBI \qquad 2 \le i \le 5 \tag{3.8}$$

The algorithm moves onto the next section by moving a full window size in the signal (there is no overlapping in this windowing technique) and using Goto's method to detect the first beat in the section. This continues until the entire signal has been examined.

The algorithm was unsuccessful for two main reasons. The first is because it relies too heavily on the tempo rate for predicting beat positions. This method does not work

properly because the tempo value is not generally an integer value and it is impossible to always store the entire set of decimal values. The beat positions will shift over time due to the integral tempo value, resulting in incorrectly detected beats that depend on the onsets computed based on an inaccurate tempo value. This problem was one of the reasons that the window size was chosen to be fairly small. By dividing the algorithm into sections and detecting the first beat in each section, we could fix the beat position if it went awry in the previous section due to an imprecise tempo rate. The second reason the algorithm was unsuccessful was because it runs extremely slowly. It takes 5.4 minutes to examine a synthetic signal that is 22 seconds in length and 13.6 minutes to analyze a musical signal that is 55 seconds. These two difficulties motivate us to develop a modification of Goto's beat detection algorithm based on our own improvements.

### 3.2.2 New Algorithm Details

The process behind beat detection is analyzing a musical signal and finding the positions of all the beats. Goto's original algorithm uses drum patterns and chord change information to make the system more robust, but we have not included these features in our implementation. Instead, we rely on Tzanetakis' tempo algorithm detailed in the previous section to give us more accuracy in determining a beat onset. Our system does not require the precision that Goto's algorithm strives for so we fashioned a simpler version of his system that runs in close to real-time and does not require additional musical knowledge.

This algorithm uses the FFT and power spectrum to build a signal in the time-frequency domain. A moving Hanning window is used to analyze the signal piece by piece, with a window size of 1024 samples and a step size of 256 samples. The Hanning window is applied to the signal by multiplying it with an equal sized portion of the signal. The FFT is applied to all samples within the window and the power spectrum is computed by multiplying the FFT result by its complex conjugate. The values of the power spectrum correspond to the power at each particular frequency. The window then moves to the next section of the signal, with a ¾ overlap with the information in the previous window due to the step size. In order to convert all this information into the time-frequency domain, it is necessary to compress the set of samples in each window

into a single frame. Goto's algorithm measures time in *frame-time*, where each frame-time corresponds to one window span, or 1024 samples. The power spectrum $p(t,f)$ is represented as the power of frequency $f$ at frame-time $t$.

The next step of the algorithm involves extracting onset components from the power spectrum. Onset components are found at frequencies where an increase of power occurs. By searching the immediate neighborhood in time and frequency space around $p(t,f)$ the *degree of onset* $d(t,f)$ is calculated. The degree of onset is the amount of power increase between the frame-times. The following segment details this step:

$$
\begin{aligned}
&pp = median(p(t-1, f-1), p(t-1, f), p(t-1, f+1)) \\
&mn = \min(p(t, f), p(t+1, f)) \\
&\text{if } mn > pp \\
&\quad d(t, f) = \max(p(t, f), p(t+1, f)) - pp \\
&\text{else} \\
&\quad d(t, f) = 0
\end{aligned}
\tag{3.9}
$$

When $pp$ is computed in Goto's implementation he uses the maximum value of the neighborhood. We choose to use the median to get a more faithful representation of the whole section. If the median of the neighborhood around the previous frame-time is smaller than the minimum power value of the current and next frame-times then the power is increasing over time and an estimated onset component exists at this position in time-frequency space.

The third step splits up the onset components into 7 frequency bands for further analysis. The frequency-time space is converted to time space by adding up all the frequency components within each frame-time $t$. Only the frequency components that fall within the current frequency band $i$ are involved in the summation.

$$D_i(t) = \sum_f d(t, f) \tag{3.10}$$

This is executed for each of the seven frequency bands and performing convolution with a Gaussian filter smoothes the result.

The three main steps of the algorithm are fairly similar to Goto's original method. The majority of the next few steps are new contributions of ours to the implementation to integrate tempo information and to make the technique robust enough for our purposes.

To narrow down the range of possible beat positions, a threshold is used to remove frame-times with the smallest amplitudes. This is based on the assumption that beat sounds are fairly high in amplitude compared to other musical features. The threshold is computed by multiplying the maximum value of each frequency band with a percentage value. The percentage value ranges from 80-90% of the signal's amplitude, meaning the values in $D_f(t)$ that fall within the highest 80-90% of the signal's amplitude are retained and the rest are discarded. It is important to note that the amplitude of the beat is dependent on the amplitude of the signal. If the dynamics of the signal at a point in time are soft, then the amplitude of the beat will be low to match this, as will the amplitudes of the other musical features. This detail is the reasoning behind the choice of the percentage value. If the percentage value does not cover the softer ranges of music then the beats are not detected in those time intervals. The percentage value that works best for the threshold changes from song to song and is manually set based on experimentation.

The estimated onset times are put into onset-time vectors for further comparison across frequency bands. For each frame-time a vector is created that denotes whether or not an onset has been found in a frequency band at that point in time. A value of 1 denotes that an onset peak is detected and a value of 0 denotes that nothing is detected. An onset rate-of-recurrence value for a frame-time is calculated by adding up the vector values across all frequency bands. A result of 0 means that an onset peak has not been detected in any frequency band at the current frame-time, while a result of 7 means that an onset peak has been detected in all frequency bands. Storing only the frame-times where an onset has been detected in more than one frequency band further narrows down the number of estimated onset times.

Goto uses a multiple-agent beat prediction method to determine the correct position for the next beat. We use a much simpler technique that utilizes the tempo information calculated by Tzanetakis' algorithm. An inter-beat-interval (IBI) is the distance between two beat onsets and can be calculated based on the tempo of the song. A direct relationship occurs between the speed of the song and the distance between beats and this relationship is used to compute the IBI directly from the tempo. The tempo is first converted from beats per minute to beats per second (bps) by dividing it by 60 (the

number of seconds in a minute) and then converted to discrete samples by the simple division of *44100/bps*. The IBI is this conversion result divided by the original step size of 256. The IBI must be in frame-times to correspond to the unit in which the beat onset positions exist.

$$\begin{aligned} bps &= bpm / 60 \\ samples &= 44100 / bps \\ IBI &= samples / stepSize \end{aligned} \tag{3.11}$$

The first estimated beat is stored as the first true onset of the signal and used as a comparison point for the next estimated beat in the list. The distance between this first actual onset and the next estimated beat is calculated. If the distance is greater than the *IBI-error*, where the error value is 5 frame-times, then it is stored as the next actual onset in the signal. The threshold check ensures that the final beat onsets are not too close together, as can be the case when the algorithm detects weak beat positions. Weak beats are the beat sounds that occur between the actual beats of a song. They are generally found at twice the tempo rate and half the distance between two actual beats and can be mistaken by beat detection algorithms as real beats. Tracking of these beats is avoided by using the IBI to ensure only beat positions that occur around or further than the known interval are chosen. Beat positions that occur further than the known interval are considered as a precaution for when the algorithm cannot detect beats around the current estimated position. The algorithm will be able to recover itself by looking further in time for the next beat while skipping the current estimated beat position. This procedure is followed for all the estimated beats in the list and the end result is a vector of actual beat onsets for the entire song. The beat positions must be converted from frame-times to frames because Maya uses the unit of frames in its animation system. The conversion equations use the step size of 256 samples, the frequency sampling rate of 44100 samples/second, represented by *Fs*, and a frame rate of 24 frames per second (*fps*).

$$\begin{aligned} beats &= onsets \cdot stepSize / Fs \\ frames &= beats \cdot fps \end{aligned} \tag{3.12}$$

The beat detection algorithm has a good accuracy rate, but tweaking of the percentage value for the beat onset threshold is necessary because the value that works best changes from one song to the next. Despite the use of the IBI to prevent weak beats from being detected, occasionally the algorithm will skip a true beat and choose the next

weak beat. This occurs when the onset vector does not find any beats around the true beat position. If the next weak beat position is greater than the distance threshold, as well as the closest estimated beat to the current one, it is chosen. This is one problem that will be addressed in future work because it has the tendency to change the detected beat structure. The algorithm is extremely efficient at recovering once it has lost the correct beat structure, usually taking no more than a couple of seconds to correctly identify the beat once again. The results of our modified version of Goto's algorithm are presented in Section 3.2.3.

### 3.2.3 Beat Detection Testing and Results

Testing of the beat detection algorithm has to be performed manually in order to assure accuracy. Both visual data and audio data are used to compare the generated results with the true beat positions in the musical signals. Visual data is used for the synthesized signals where the beat positions are discernable. Audio data is used for all other signals where it is easier to hear the beat than find it in the signal.

Simple testing was performed first on synthesized audio signals that are composed of beeps that represent the beat. The purpose of the simple synthesized signal is to determine how well the beat detection algorithm can find the beats of a given tempo. A user-determined tempo value is utilized to establish the IBI and short beeps are added to the synthesized signal at positions derived from the previous beat position plus the IBI. To make the synthesized signal more similar to real musical signals, beeps are added at the quarter note level, the half note level and the whole note level. A single beep at quarter note positions is not enough to synthesize a song similar to real music because there are generally many instruments playing, which increases the amplitude of the beats. Adding beeps at 1,2 and 4 times the frequency will increase the amplitude at those points in a similar way to multiple instruments playing on the beats.

White noise is added into the signal through the use of a Matlab function named *randn*. This function creates vectors and matrices filled with random numbers built from a normalized distribution. The amplitude of the resulting noise signal is normalized and then divided by a specific number, *divisionNumber*, to reduce the amount of noise. The beeps are added into the noise signal to create the synthesized result.

$$s = randn(1000000,1)$$
$$mx = \max(s)$$
$$signal = (s / mx) / divisionNumber \quad (3.13)$$

Different amounts of noise are added to each synthesized signal to determine how much the noise level affects the beat detection algorithm. The noise's division value varies from 100 (small amount of noise) to 2 (intermediate amount of noise). The signal is also tested with no noise at all. The new algorithm can detect only a small amount of beats when the division value is 2, so testing was stopped at this point.

| Noise Level | Old Algorithm | New Algorithm |
|---|---|---|
| none | 28/31 | 31/31 |
| 1/100 | 31/31 | 31/31 |
| 1/50 | 31/31 | 31/31 |
| 1/25 | 31/31 | 31/31 |
| 1/16 | 31/31 | 31/31 |
| 1/8 | 31/31 | 31/31 |
| 1/6 | 31/31 | 31/31* |
| 1/4 | 31/31 | 31/31** |
| 1/2 | 20/31 | 12/31*** |

Table 3.1: Comparison of the old and new beat detection algorithms using a synthetic signal with a tempo of 82 bpm. The audio signal is 22 seconds in length, with a total of 31 beats. Testing is cut off at a noise level of ½ the beep's amplitude because this is where the new algorithm fails considerably. The second and third columns denote the number of beats out of 31 detected correctly by each algorithm.

Table 3.1 shows the noise level affecting both beat detection algorithms around the same time. The signal with a noise level of ½ affects both algorithms, with the old algorithm detecting eight more beats than the new one. The asterisk symbols in the New Algorithm column of the table denote that different threshold values were used for testing those three particular signals than for testing of the previous six signals. The six signals with the smallest signal-to-noise ratio work perfectly with a threshold of 90%, while the seventh signal (*) uses a threshold of 72%, the eighth signal (**) uses a threshold of 45% and the ninth signal (***) uses a threshold of 35%. The new algorithm only detects 5 beats in each of these signals if the threshold value is left at 90%. Although the new algorithm detects only 12/31 beats in the last synthesized signal, Figure 3.3 demonstrates that the detected positions are not far from the actual beat positions.

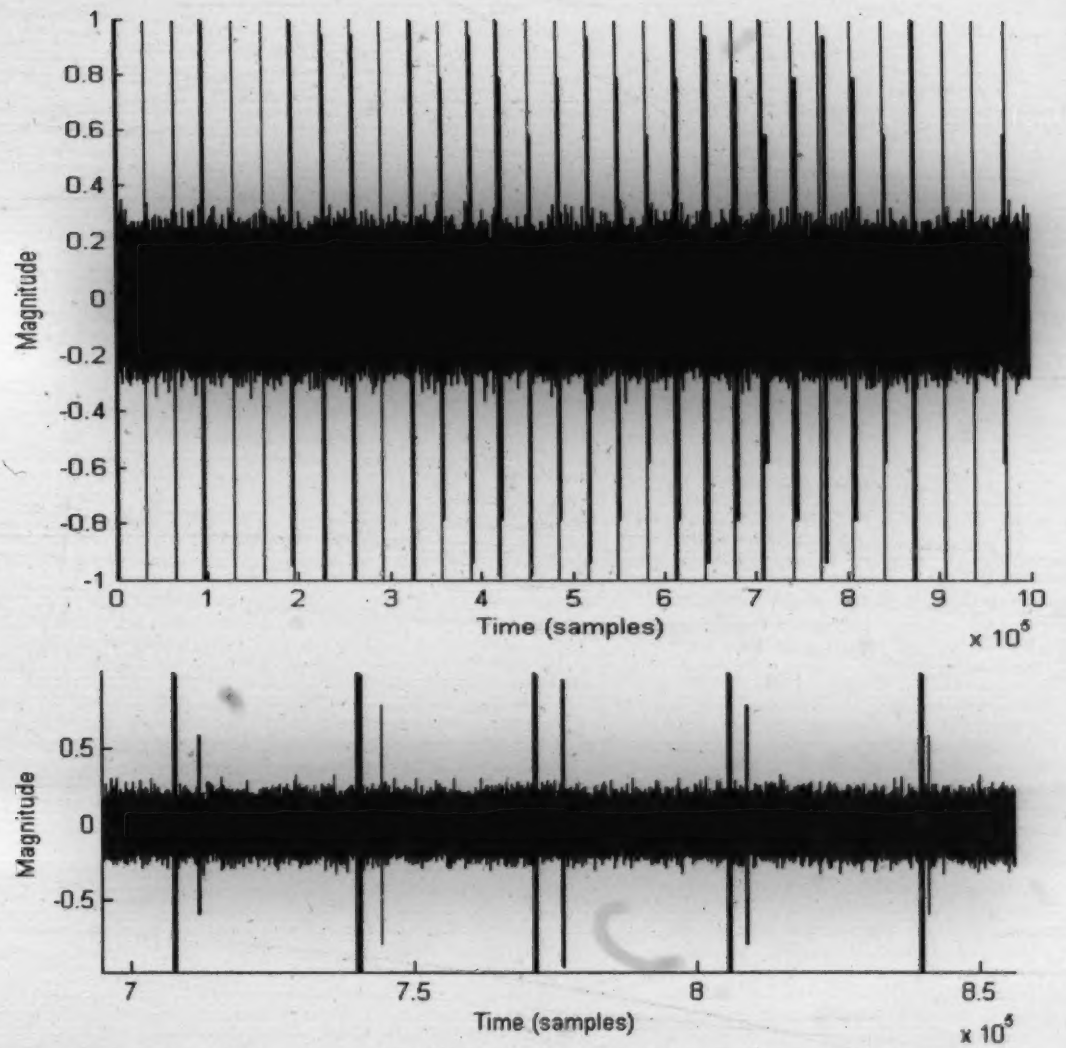



Figure 3.3: The results of the new beat detection algorithm on a synthesized signal with a tempo of 82 bpm. The noise level is 1/2 and a 35% threshold value is used. Blue lines represent the original beats in the signal and red lines the beat positions detected by the algorithm. The top image displays the full signal after beat detection, while the bottom image a closer view of the distance between actual beats and detected ones.

Table 3.2 displays the results for testing the old beat detection algorithm on ten synthetic signals with different random seeds and a tempo of 153.3682 bpm. The values in each column represent the number of beats detected for each signal with a specific noise level, where the noise level is denoted in the second row of the table. This testing shows that the random seed does not affect the developing trend of the noise level

| Signal # | Number of beats detected (/58) for each noise level | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | *1/100* | *1/50* | *1/25* | *1/16* | *1/8* | *1/6* | *1/4* | *1/2* |
| 1 | 35 | 45 | 40 | 45 | 35 | 43 | 40 | 40 |
| 2 | 45 | 45 | 41 | 35 | 48 | 37 | 42 | 30 |
| 3 | 45 | 35 | 35 | 35 | 36 | 43 | 35 | 22 |
| 4 | 38 | 40 | 35 | 40 | 40 | 35 | 35 | 32 |
| 5 | 35 | 35 | 38 | 35 | 35 | 37 | 35 | 31 |
| 6 | 35 | 35 | 35 | 35 | 35 | 45 | 35 | 33 |
| 7 | 35 | 39 | 39 | 39 | 35 | 35 | 39 | 38 |
| 8 | 38 | 35 | 35 | 35 | 45 | 40 | 35 | 36 |
| 9 | 40 | 35 | 35 | 40 | 35 | 35 | 36 | 41 |
| 10 | 35 | 35 | 35 | 35 | 35 | 35 | 40 | 36 |
| **Average beats detected** | 38.1 **38** | 37.9 **38** | 36.8 **37** | 37.4 **37** | 37.9 **38** | 38.5 **39** | 37.2 **37** | 33.9 **34** |

Table 3.2: The results of using the old beat detection algorithm on 10 synthetic signals with random seeds and a tempo of 153.3682 bpm. Eight different noise levels are used, ranging from 1/100$^{th}$ of the beep amplitude to ½ of the beep amplitude.

influencing the accuracy of the algorithm. The average number of beats detected is fairly even until the noise level reaches ½ of the beep amplitude, at which the result dips.

Table 3.3 displays the results for testing the new beat detection algorithm on the ten synthetic signals used in Table 3.2. The algorithm uses a threshold value of 90% to obtain the majority of the results seen in Table 3.3. The asterisk symbols in the noise level row (the second row) denote that different threshold values were used to obtain these results than the threshold used in the first six columns. The threshold used by the algorithm in column 7 (*) is 72% while the threshold used in column 8 (**) is 65% and the threshold for column 9 (***) is 25%. This indicates that the new algorithm is quite robust because its threshold value can be altered to reflect the signal. The results are consistently excellent until the last noise level is reached, at which they drop off considerably. This shows that the random seed does not affect the reliability of the new beat detection algorithm. From the comparison of the values in Table 3.2 and Table 3.3, it is evident that the new beat detection algorithm performs considerably better than the old beat detection algorithm. In the majority of cases it detects 100% of the beats, while the old algorithm detects 67% of the beats in the best case.

| Signal # | Number of beats detected (/58) for each noise level | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | *1/100* | *1/50* | *1/25* | *1/16* | *1/8* | *1/6** | *1/4*** | *1/2**** |
| 1 | 58 | 58 | 58 | 58 | 58 | 58 | 58 | 27 |
| 2 | 58 | 58 | 58 | 58 | 58 | 58 | 58 | 22 |
| 3 | 58 | 58 | 58 | 58 | 58 | 58 | 58 | 18 |
| 4 | 58 | 58 | 58 | 58 | 58 | 58 | 58 | 22 |
| 5 | 58 | 58 | 58 | 58 | 58 | 58 | 58 | 17 |
| 6 | 58 | 58 | 58 | 58 | 58 | 58 | 57 | 11 |
| 7 | 58 | 58 | 58 | 58 | 58 | 58 | 58 | 19 |
| 8 | 58 | 58 | 58 | 58 | 58 | 58 | 58 | 17 |
| 9 | 58 | 58 | 58 | 58 | 58 | 58 | 58 | 19 |
| 10 | 58 | 58 | 58 | 58 | 58 | 58 | 58 | 16 |
| **Average Beats Detected** | 58 **58** | 58 **58** | 58 **58** | 58 **58** | 58 **58** | 58 **58** | 57.9 **58** | 18.8 **19** |

Table 3.3: The results from performing beat detection with the new beat detection algorithm on ten synthetic signals with a tempo of 153.3682. Each signal was created with a different random seed and eight noise levels were used, ranging from 1/100 to ½ of the beep's amplitude.

The old beat detection algorithm has problems with this synthetic signal because the tempo is not an integer value. As displayed in Figure 3.4, the algorithm eventually loses the beat pattern because it is using a detected tempo of exactly 153 bpm. The bottom graph displays how far apart the detected beats are from the actual beats, showing the inaccuracy of the old beat detection algorithm when the tempo is not an integer. The old beat detection method does not take into account the small change in beat position that occurs due to the decimal places in the actual tempo value. It uses the detected tempo value to predict the next beat position, and in this case, the detected tempo value is not precise enough to give accurate results. This is the major difficulty with the old algorithm, which led to the development of the new beat detection method.

Once real audio data is introduced to the beat detection algorithm it is extremely difficult to visually determine where the beats occur in some songs. Synthesized beep sounds are added to the original audio signal at the positions detected by the algorithm. It is then possible for one to listen to the audio signal and compare the timing of the estimated beeps with the timing of the actual beats of the song. This testing is based on the listener's perception of the beats and our tester has an extensive musical background,

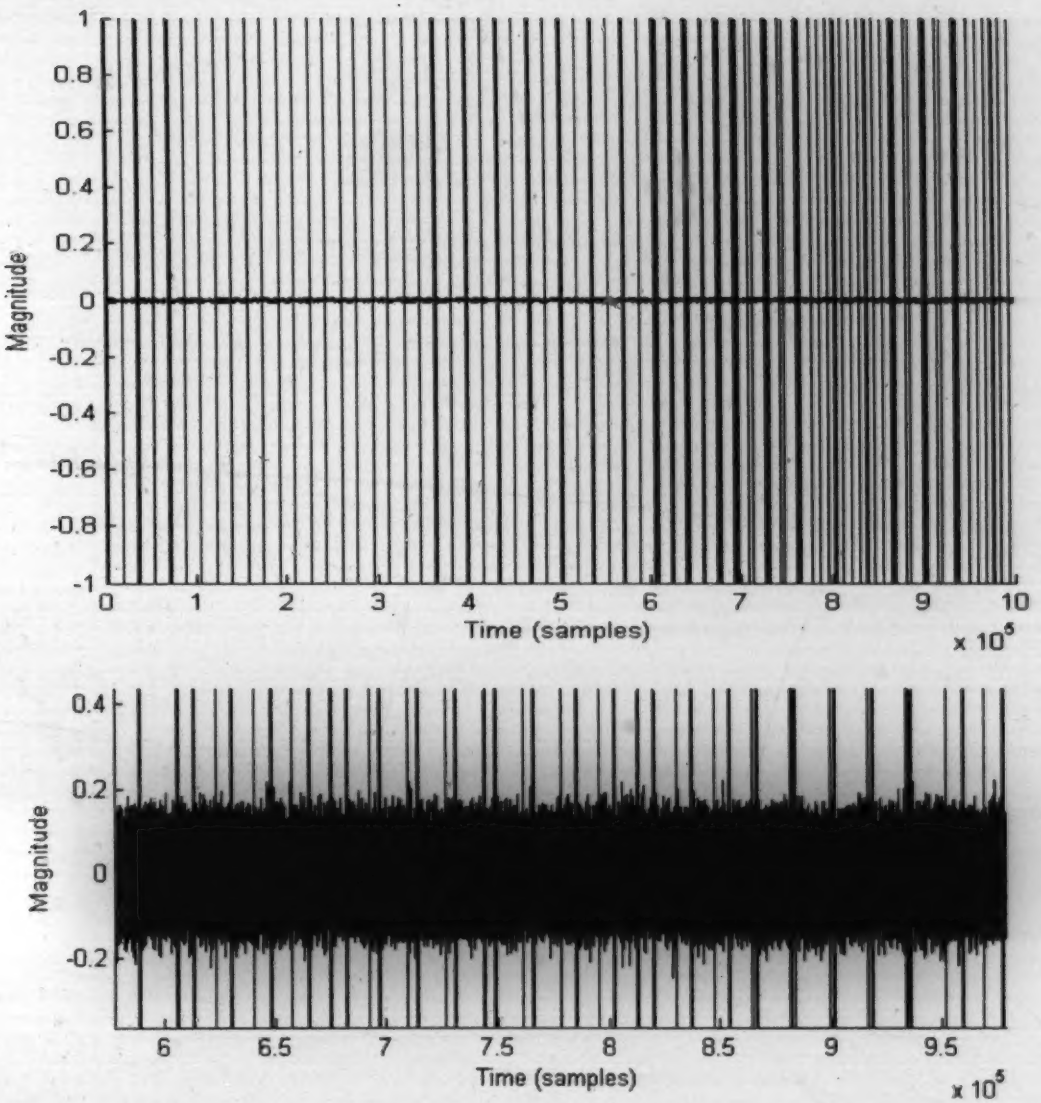



Figure 3.4: The results of the old beat detection algorithm on a synthesized signal with a tempo of 153.3682 bpm. The noise level is 1/100. Blue lines represent the original beats in the signal and red lines represent the beat positions detected by the algorithm.

making it easy for her to accurately compare estimated beat positions with actual beat positions. Similar to the procedure applied for synthesized signals, visual data is acquired by graphing the original signal in one colour and graphing the beeps corresponding to the estimated beats in another colour. In some cases it is easy to verify if an estimated beat has been correctly placed, while other cases rely more on the audio

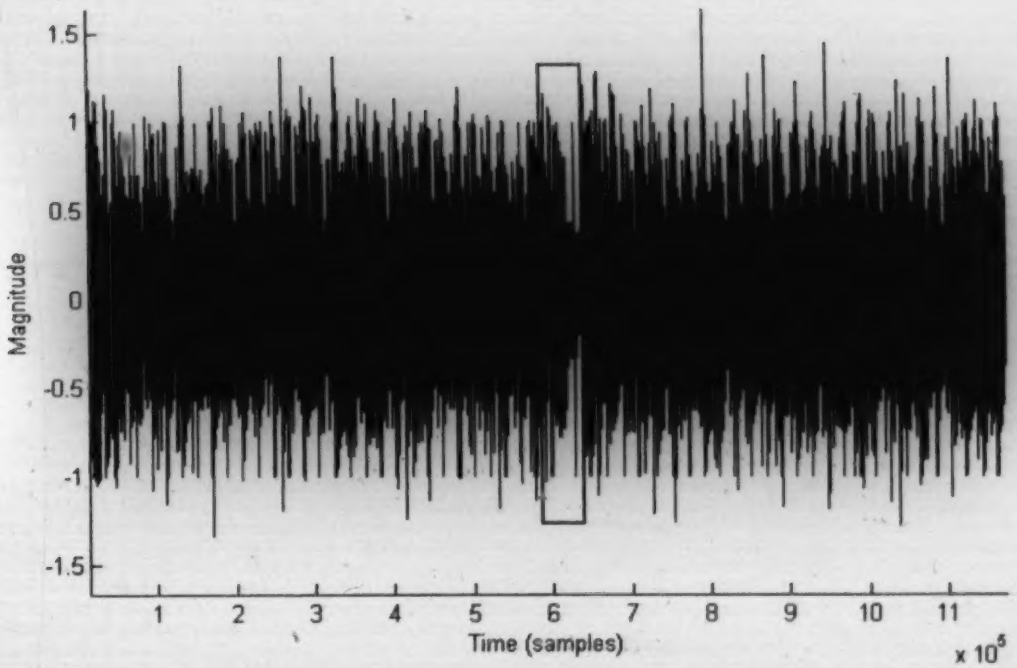



Figure 3.5: A signal graph of the Celtic song "Warriors" used by the Celtic animation system. The tempo of this song is 135 bpm. The blue lines symbolize the original signal and the red lines represent the detected beats. One can see a space where a beat should occur around the 65,000-sample position, as marked by the green square. The algorithm is able to recover from this missing beat and rediscover the beat structure immediately.

data than the visual data. The visual data also helps us determine where problems occur in the algorithm along with possible types, such as if the algorithm has missed a beat or if the beat structure has gone awry. Some of these visual results for the new beat detection algorithm are displayed in Figures 3.5 and 3.6.

## 3.3 Dynamics Extraction Algorithm

The dynamics of a musical piece are one of the few musical features that can change the expression and mood of a song without actually changing the structure of the song itself. Dynamics consist of the louds and softs of the music, including transitions between the two that also known as crescendos (soft to loud) and decrescendos (loud to soft). The dynamics levels are extracted because they are useful in creating corresponding movement. Dance moves take into account dynamics and base the strength of a

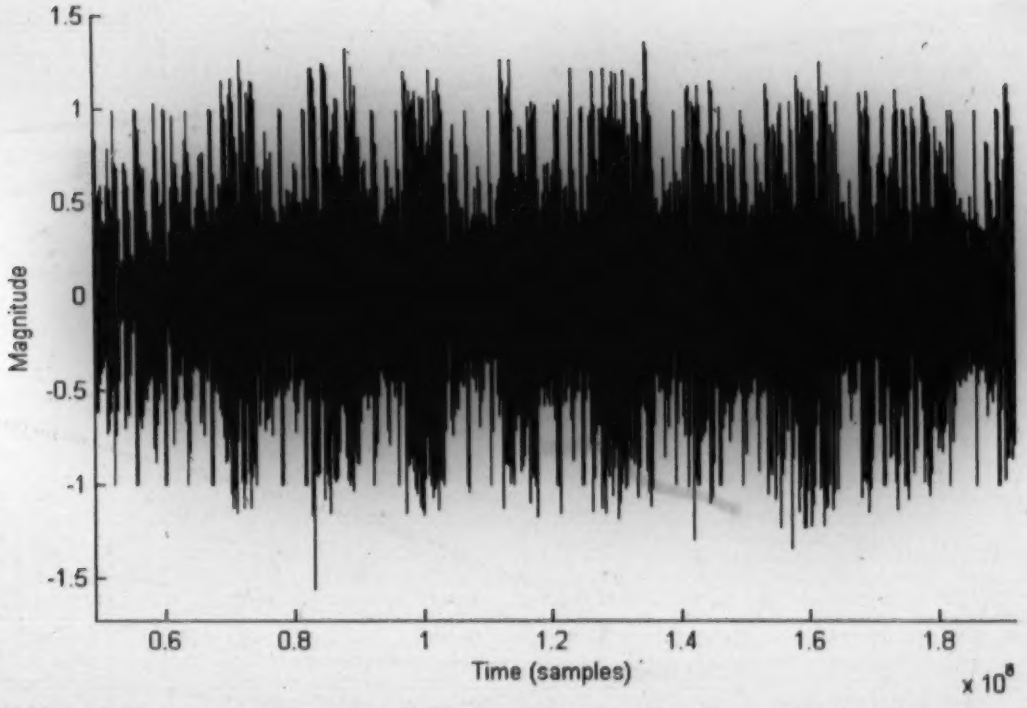



Figure 3.6: A graph of a segment of the musical signal from the rock song "Brown Eyed Girl" by Van Morrison. The tempo of this song is 76 bpm and it is significantly slower than the Celtic song displayed above in Figure 3.5. The blue lines symbolize the original signal and the red lines represent the detected beats. The intervals between detected beats are noticeably larger in this signal because of the slower tempo.

movement on the strength of the song at a point in time. The purpose of this extraction algorithm is to detect the 50 positions where the dynamic level is highest and 50 positions here the dynamic level is lowest. These positions represent the loud and soft dynamics respectively.

Dynamics are extracted by using a moving window with a size of 44100 samples to compute the power spectrum of the music signal. . The FFT is performed on the information in each window and the result is multiplied by its complex conjugate. The inverse FFT is performed on the outcome. Since the signal is symmetric, the second half of the signal is removed and the algorithm proceeds to calculate the absolute values for the signal's first half. The maximum and minimum values are located and added to a list before the window is moved. This technique is performed for each window until the

entire signal has been analyzed, with the resulting list being comprised of the highest and lowest values from each window.

Finally, the algorithm determines the 50 highest and lowest values in the temporary list and stores them as the dynamic positions. The system detects 50 of the highest and 50 of the lowest values because we believe that 100 dynamic positions are enough to build a complete representation of the dynamic structure of the song. These positions are converted to frames by using Equation 3.12. Crescendos and decrescendos can also be represented by the dynamic positions. A transition over time from a high dynamic value to a low dynamic value signifies a decrescendo while a transition from a low dynamic value to a high dynamic value signifies a crescendo. This information can also be used in the animation system to help the motion better express the music.

# Chapter 4

## Hip-Hop System

Dancing is an art form that invites creativity, originality, imagination and inspiration to its creation process. Dancing styles range from the structured dances found in ballroom dancing to the chaos that exists in hip-hop dancing. One of the goals of the system detailed in this thesis is to combine different music and dance styles and observe how they interact with each other. The ability to experiment in this way is a fairly novel idea that cannot be easily done with synchronization-based systems. The movements are built based on beat onsets and musical dynamics expression. These motions reflect a specific dance style but their timing and strength is completely based on the corresponding music file. The resulting animation is a unique representation of a dance style that is specifically tailored to a piece of music.

The hip-hop system is a prototype system for people to experiment with different ways of arranging music and movement. The hip-hop dance style was chosen because it is extremely interesting and unique in its composition. By viewing different dance videos, it is also fairly apparent that complex dance sequences are made of smaller primitive movements, which lends itself well to one of the main purposes of this system. We are looking to show that primitive movements can be combined to create complex dance sequences. Hip-hop is not a structured dance so it gives the user extreme freedom to arrange the movements the way she wants without constraints. The system automatically interpolates between all movements, which removes the responsibility from the user to create smooth transitions between primitives.

This chapter presents the major components of the Hip-Hop system. Descriptions of all the primitive movements implemented in the system are included, along with explanations of the different mapping processes between music attributes and movements. The creation and parsing of the script file is detailed and the purpose of the random function is addressed. Lastly, problems of the system are described.

## 4.1 System Overview

The Hip-Hop system is built from two key pieces: music analysis and motion synthesis. The motion synthesis section involves interacting with the user, creating motion and coordinating movements and music. This process involves script files, mappings and primitive movements. These components control the animation and continuously interact with one another to produce the final motion sequence.

One of the main goals of this system is to give the user as much control as possible over the final animation without her having to build the movements herself. This is achieved through a text file known as a script file. This file allows the user to specify the movements that compose the animation. The user communicates her commands to the system through the script file by identifying the movements that the character will perform as well as the timing of the motions. Movement timing is controlled through a process called mapping.

The mapping process is designed to allow the user to choose when certain movements will occur and which body parts will perform the movements. Its purpose is to ensure the user has control over the final result, giving her the opportunity to design her own dance from scratch using various body parts, movements and musical attributes. In many cases the user of the system will be inexperienced in animation and will be unsure of the best way to set up the movement sequence and its subsequent mappings between movements and music. A random function has been implemented that will randomly choose the order of a set of movements for the user.

Primitive movements are simple movements that are used in combination to create more interesting and complex motion. Several types of primitives are implemented in this system and they can be used on any body part and at any point in time. Some primitives work well with specific body parts, but the user is encouraged to experiment with different groupings. The following sections provide more details on the motion synthesis component of the animation system, including information on primitive movements, mappings and script files.

## 4.2 Script Files

The user designs the animation through text files known as script files. They allow the user to set up the order in which the movements are performed as well as all the mappings between body parts, primitive motions and musical attributes. The script files provide maximum control over the final result because they supply the system with all the information it needs to build a motion sequence. The system has a special parser that reads the script file and inputs its details into the animation component. The animation component builds the motion sequence from the timing and movement information entered by the user. This section discusses both the design of a script file and the parser that inputs the corresponding information into the system.

### 4.2.1 Designing Script Files

The script file is designed so that the user can easily incorporate different combinations of primitive movements and mappings into the animation, choose the time periods for the mappings and use a character of her choice in the final result. Unlike other animations systems such as Maya and 3D Studio Max, the user does not have to position the character or set keyframes for movement. The script file provides the user with the means for commanding the system to do these things for her.

The script file is set up so that the user is able to interchange the character used in the animation. The script allows the user to call the various body parts of the character rig by any term, rather than forcing the user to adhere to a strict naming policy. The system proceeds to find the body part specified in the scene and move it according to directions. Unlike the body parts, the user cannot change the names of the primitive movements. These motions are given specific names by the animator during implementation and the system will not recognize any other term. A mapping between a body part and a primitive movement occurs by specifying first the name of the primitive movement and then the name of the body part in the scene. The format is as follows:

MOVEMENT: bodyPartName

Examples:

- SWAY: UpperBody
- LIFT: RightArmCtrl

Multiple characters can be used through this mapping method simply by incorporating their body part names into the script file. The system is unable to determine the difference between characters because it only selects what is specified in the script file. If the user chooses to use body part names that range across several characters then those body parts will be moved regardless of whether they belong to the same character or not. This procedure allows a user to move multiple characters in the same way she would move a single character.

Two of the primitive movements perform special motions that require an object in the scene. The Drum movement requires the name of the drum object that the character will be drumming on and the Throw movement requires the name of the object the character will be throwing in the air:

- DRUM: BodyPartName, DrumName;
- THROW: BodyPartName, ObjectName;

These movements need special consideration when creating the script file because the system takes into account both the body part and the extra scene object when creating the mapping. The object must be listed in the script file after the body part so that the parser can tell the difference between the two components of the movement and perform the correct motion.

Once the user has determined which body parts will be performing which movements, she needs to decide when these movements will occur in the animation. The timing is chosen through the primitive movement to musical attribute mappings. The musical features that this system can map to are listed below. The labels used to call them in the script file are found in parentheses behind the musical attribute term.

- Strong beat (STRONG_BEAT)
- Weak beat (WEAK_BEAT)
- Loud dynamics (LOUD_DYNAMICS)
- Soft dynamics (SOFT_DYNAMICS)

The format for defining the mapping in the script file is as follows:

MUSICAL_ATTRIBUTE
MOVEMENT: BodyPartName,
MOVEMENT2: BodyPartName2;

The user can list as many movements under a musical attribute as she wants. Each movement must be mapped to a body part however, and no body part name can occur under the same musical attribute more than once. This constraint is to prevent the character from attempting to perform two different movements with the same body part at the same time.

An animation where the same movements are being performed for its entire duration is mundane and uninteresting. The script file design allows the user to divide the animation into smaller time periods and produce different mappings at each interval. The user specifies the start frame and the corresponding mappings for each interval. This is repeated for the number of intervals chosen by the user. The length of each interval depends on the user, with an interval continuing until the next specified start frame has been reached. The *START* keyword designates the beginning of each interval, with the start frame number appearing directly after the term. The movements, music attributes and body parts used in the animation can be changed from interval to interval, as well as the manner in which they are mapped to each other. This creates a more interesting animation where the character's movements change over time. An example of a script file is found in Figure 4.1.

### 4.2.2 Parsing Script Files

The script file can hold a large amount of information about an animation, including numerous mappings and interval sections. This information must be handled by the system in the proper way or else the resulting motion sequence will not follow the user's specifications. A special parser was implemented in order to properly organize the information retrieved from the script file. This parser hunts for keywords in the script file and uses these keywords to build up structures that hold mappings and maintain their proper timing. It is the job of the parser to divide up the script information.

The five keywords that the parser searches for in the script file are START, STRONG_BEAT, WEAK_BEAT, LOUD_DYNAMICS and SOFT_DYNAMICS. As it retrieves each word in the script file in order, it checks if the new word matches one of the musical attribute keywords. If a match occurs, the parser proceeds to recover all the mappings corresponding to this particular keyword and store them in a specialized

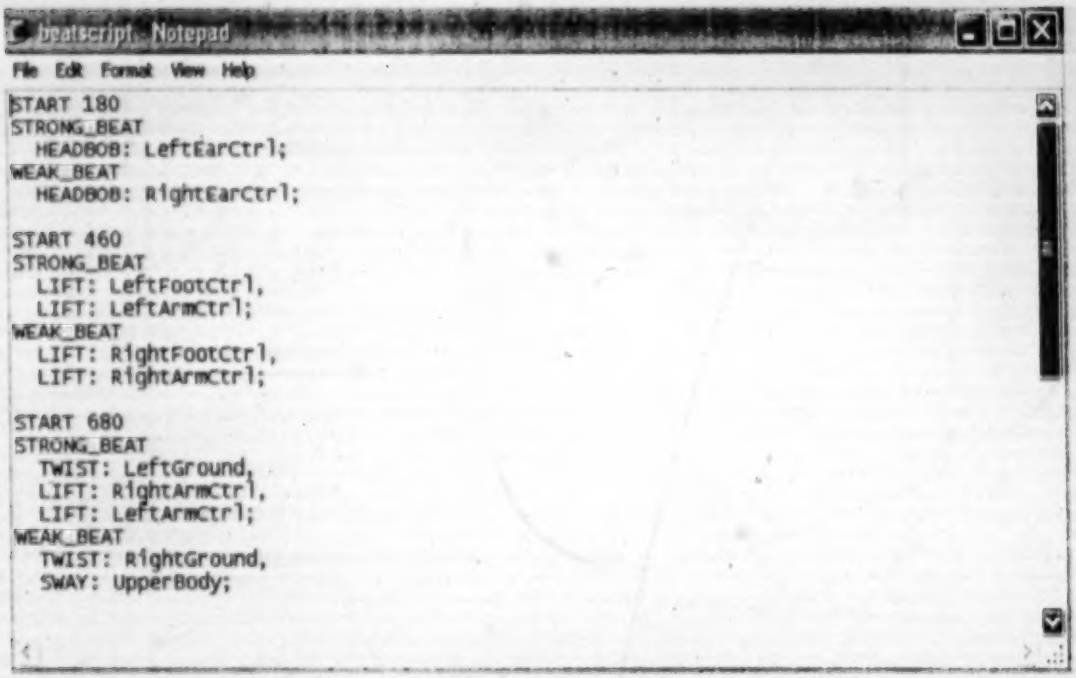



Figure 4.1: An example of a script file segment for the Hip-Hop animation system. The mappings between body parts in the scene and primitive movements are defined, along with the mappings between movements and musical attributes. The user can create different intervals of movements to make the animation more interesting.

structure. The punctuation used in the script file is extremely important is this process. Punctuation symbols are used to divide up the information in the script file. A colon (':') character symbolizes the mapping between a primitive movement and a body part in the scene. When the parser finds this character it knows that the word before the colon corresponds to the name of a primitive movement and the word after it corresponds to a body part. The comma (',') is found at the end of a body part name and symbolizes that another movement-body part mapping follows the present one under the current musical attribute. A semicolon (';') represents the end of the list of movement-body part mappings that are used when the current musical attribute occurs. When the parser discovers this character, it knows that it has finished with this particular musical attribute and that the next set of mappings will be for a new one. The parser continuously retrieves data using this approach until the end of the file has been reached. A full example of how punctuation is used can be seen in Figure 4.2 below.

There are three special keywords that do not correspond to musical attributes, but that the parser will still recognize. These keywords are: DRUM_NAME, DRUM_TYPE and UPDOWNTYPE. The user specifies these terms after all the mappings for a time interval have been identified. The drumming keywords contain information necessary for the execution of the Drum primitive. The DRUM_NAME keyword indicates the name of the drum object in the scene while the DRUM_TYPE keyword stipulates whether the drumming movement between two arms will occur on the same beat or on alternate beats. The UPDOWNTYPE keyword is used for the Updown primitive movement and, like the DRUM_TYPE keyword, it identifies if the Updown movement between two arms will occur on the same beat or on alternate beats. No punctuation is required for these keywords. An example on their use is found in Figure 4.2.

Figure 4.2: Keywords and punctuation are extremely important to the parser when gathering information from the script file. The parser relies on both features to divide up the script details into their proper structures for later use by the animation system.

While the parser is effective at dividing up the information into its proper structures, it is not particularly robust. If the wrong punctuation is placed somewhere in the script file the parser is not intelligent enough to decipher the user's mistake. Misplaced punctuation affects the way the rest of the script file is read, which results in an incorrect

final animation. The responsibility to check punctuation currently lies with the user and this can lead to frustration. The parser was designed this way because it is difficult to attempt to understand the user's intent and then proceed to correct it. It is more frustrating if the parser misunderstands the objective and changes the script file incorrectly than if the user is left to make the corrections on her own.

## 4.3 Mappings

In order for the system to create an animation tailored to the music, it must have knowledge of how the music affects the movements. This knowledge is provided by the user through two different types of mapping. The first type of mapping is between body parts and primitives, where the user chooses the body part that will perform a specific primitive movement for a period of time. The second type of mapping is between primitives and musical attributes. For each musical attribute the user can choose the movement that will be performed when the attribute occurs in the music. This setup encourages the user to experiment with different mapping combinations. Mappings are the basis for the creation of the animation because they define what movements will occur, when they will occur, and how they will be used by the system.

There are four musical attributes that are utilized by the system for mapping purposes: strong beat, weak beat, loud dynamics and soft dynamics. The strong beats are the pulses in the music that a listener taps her foot along with, while the weak beats are the pulses in between the strong beats. In most pieces of popular music the beat is in 4/4 time. The strong beat occurs on the first and third beat, while the weak beat occurs on the second and fourth beat.

The majority of the primitive movements work well when performed by any body part. The animation system allows for the same movement to be repeated throughout the animation at different times and by different body parts. For example, both of the character's arms can throw an object into the air at the same time. This encourages the user to take advantage of movements that work well in the animation by repeating them in different contexts. Each movement must be mapped to a body part in order for it to be executed. There are no constraints on this mapping, so it is at the discretion of the user when choosing primitive-body part combinations. The same movement can be mapped

to more than one body part, but each body part can only be mapped to one movement in each time interval. In essence, this means that a body part is not restricted to a single movement over time. For example, a body part can be mapped to one movement when the strong beat occurs and then mapped to a different movement when the weak beat occurs because each beat takes place in a different time interval.

Mappings between primitives and musical attributes are not limited to one-to-one relationships. Several primitives can be mapped to the same musical attribute. For example, the movements Sway, Updown and Headbob can all occur on the strong beat. In the same way, several musical attributes can use the same movement. For example, the Lift primitive can be performed when the strong beat and the weak beat occur. A large number of mapping combinations exist, which results in the ability to create numerous animations from the same set of movements. The user can take a set of primitives and map them to the musical attributes. She can then take the same set of primitives and simply change which musical attributes each one maps to in order to produce a completely different animation. This type of mapping changes the time period over which a movement occurs, resulting in a different representation of the music by the character.

The script file ensures that the user has extensive control over the final animation, but difficulties can occur if the user is unsure of when movements should be performed or if she cannot formulate good mapping combinations. A random function was implemented to resolve this issue. The purpose of the random function is to give the user control over the movements that will occur, but remove the difficulty of deciding their timing. It provides the user with a unique outlook on how the animation would look with different movement groupings and is an easy way for her to experiment with mappings.

The random function takes a number of movement-body part pairs as input and randomly chooses when each one will be used in the chosen time interval. The user specifies the candidate movements in the script file, just as she would do normally, and indicates that she wants to use the random function by indicating the keyword after the musical attribute it applies to:

MUSICAL_ATTRIBUTE RANDOM

The random function will first divide the interval into smaller subintervals. The number of subintervals corresponds to the number of candidate movements, and the length of each subinterval is chosen randomly. The movements that will occur in each interval are then chosen randomly by the system and can be chosen more than once. Only one movement will be selected for each subinterval. An advantage of using the random function is that it is not predisposed to choose one movement over another, and therefore it can create unique combinations that the user might not otherwise discover on her own. The random function is called each time the system is restarted, with previous sequences being removed, so the user can create several different animations from the same script file.

## 4.4 Primitive Movements

It is our belief that complex movements can be decomposed into small primitive moves. The majority of primitive movements can be used on any body part of the character, with the main body parts utilized being the head, torso, arms, and legs. In the case of the bunny figure displayed in the Results section, the ears can also move. The primitives can be employed in any order by the user to create different combinations. The ability to use the same primitive on more than one body part allows for a larger number of possible movement combinations for the character at a point in time. The difficulty lies in creating combinations that work well together and this is the responsibility of the user. A total of ten major primitive movements are included in the system, with an additional nine primitives being used to create some of the major ones. The user cannot call the minor primitives in the script file. The minor primitives and their purposes make up the first set of movements listed below, with the major primitives discussed second:

Minor Primitives:

- **AwayDrum** – moves a body part, usually an arm, away from the drum. This movement generally occurs directly after the character has hit the drum and is the reaction to the original movement. It is the second part of the Drum movement.

- **DownHead** – used by the HeadBob primitive, it rotates the head downwards so it is looking at the ground.

- **LimbDown** – once the character has released the object being thrown in the ThrowObject primitive, this movement brings the arm back down to its original position to catch the object. It is used to create a realistic throw and catch movement by giving the appearance that the arm is reacting to the momentum of the object by jerking downwards slightly as the object is caught.

- **LimbUp** – moves the character's arm upwards in a throwing motion. It is used in the ThrowObject primitive to create a realistic release motion.

- **TapDown** – this movement brings the body part, such as the foot, back down to the ground after it has been rotated upwards. The original position has the foot on its heel with the toes in the air. This movement is the finishing move of the Tap primitive, which is meant to emulate a foot tapping to the beat.

- **TapUp** – the beginning of the Tap primitive, this motion rotates the body part, such as the foot, upwards so that the toes and ball of the foot are in the air, resulting in the foot balancing on its heel.

- **Throw** – this movement is used only for moving the object being thrown in the ThrowObject primitive. It calculates the flight velocity necessary for the object to travel to a certain height and takes into account gravity when computing the distance traveled at each frame. It also rotates the object so that it will perform 1-2 spins in the air, just as an object being thrown in real life would. This works especially well when drumsticks are thrown up.

- **TowardsDrum** – the beginning of the Drum primitive, this movement uses the position of the drum in the scene to move the arm towards it. The purpose is to hit the drum with a set of drumsticks, so the distance to the drum takes into account the distance between the hand and the drumsticks in the hand, since we want the drumsticks and not the hand to hit the drum.

- **UpHead** – rotates the body part back up from an orientation that is facing the ground. This movement finishes the HeadBob primitive.

Major Primitives:

- **Bend** – a primitive motion that rotates a body part forward and down on one beat and then back and up again on the next beat. An example is bending at the waist to face the ground and then straightening back up.

- **Drum** – one of the most complicated motions, this primitive uses the arms to beat on a drum from the left and right sides. It needs the world position of the drum in the scene in order to compute the distance each arm needs to travel in order to hit the drum correctly. Body movement is incorporated into this primitive to convey the expression better. If a desired position is unreachable by the arm, the body

raises itself up in order to attain the position. Two different types of drumming can be achieved with this movement. The user can choose *alternate* drumming, where the arms take turns hitting the drum one at a time, or *same* drumming, where the arms hit the drum at the same time. The arm moves towards the drum and hits it on one beat and moves away from the drum to the original position on the next beat.

- **HeadBob** – this primitive can be used on more than just the character's head, but its best example of use is creating a head bobbing motion in which the body part rotates a small amount forward on one beat and then back to the original position on the second beat. It differs from the Bend primitive because the bend primitive always rotates a much larger amount as it tries to emulate the bending forward at the waist motion. This primitive creates a small head bob motion.

- **Jump** – this primitive causes the character to bend at the knees, jump in the air, bend at the knees upon landing and straighten up. It is generally paired with the torso and looks to create a realistic jump of average height over a single time interval.

- **Lift** – lifts a body part by a variable height and lowers it to the original height. The motion of this primitive occurs in one time interval.

- **Sway** – this primitive works best when used by the torso or the head. It rotates the body part so it juts out and up on one side and then swings to jut out on the other. It is meant to emulate the swaying motion of the hips. A sway from one side to the next occurs over one interval.

- **Tap** – a simple primitive that rotates a body part upwards around the x-axis in one beat and back down to the original orientation in the next beat. Its purpose is best compared to the character tapping her foot to the beat.

- **ThrowObject** – one of the most complicated primitives, this motion involves tossing an object directly upwards in the air. The object can be anything in the scene and it will be thrown up from the character's hand, perform some turns as it reaches its highest point, and be caught by the same hand. To create a realistic throwing movement, the character's arm will move down along the y-axis slightly before swiftly move upwards to toss the object. Once the object has reached the hand again, the arm will move downwards slightly along the y-axis to give the appearance of cushioning the impact of the object. This is all performed in one interval.

- **Twist** – this primitive rotates the body part around the y-axis in a twisting motion. The orientation of the body part will change by a bearing of 30° before rotating back to its original orientation, all in one interval.

- **UpDown** – moves a body part up to a specific height in one beat and down to a specific height in the second beat. The body part is constantly alternating between a high and a low height. This movement can be coordinated between two body parts, where the user can choose if the movement will *alternate* or be the *same*. If the movement alternates then one body part will move upwards as the other is moving downwards, while if the movement is the same then both body parts will rise and lower at the same time.

Dance movements should convey the expression of the music. Loud dynamics should result in grand gestures and soft dynamics should result in subtle gestures. The quality of the movement increases as its representation of the music becomes more apparent to the viewer. The hip-hop movements implemented in this system are built to change as the dynamic values change. A dynamic scale is incorporated into the animation system where the values range from 1-5. A value of 1 denotes a soft dynamic, while a value of 5 denotes a loud dynamic. The dynamic rate is computed at each frame and used to build proper expression of the music in the primitive movements. The distance that a movement travels is based on the current dynamic rate at the beginning of the movement. Dynamic rates do not change while a movement is in the middle of its execution because it results in choppy movements that switch positions in mid-movement due to the final position for the movement varying at each frame.

Primitives are implemented in one of two ways. The first is through an ease-in-ease-out function calculated using the following two equations:

$$u = (curr_time - time_start)/(time_end - time_start)$$
$$ease = (\sin(u * \pi - \pi/2) + 1)/2 \qquad (4.1)$$

where *curr_time* is the current frame, *time_start* is the starting frame for this interval and *time_end* is the ending frame for the interval. The ease value is used to compute the current translation or rotation value in the following way:

$$pos = pos + ease * (end_pos - pos) \qquad (4.2)$$

An example of a function that uses the ease-in-ease-out function is the secondary primitive downHead, detailed in Figure 4.3.

**"DownHead" Secondary Primitive Movement**

**Input:**
**f** ← current frame number
**Ri** ← 3D rotation vector for the body part at the beginning of the movement
**ts** ← starting frame of current beat interval
**te** ← ending frame of current beat interval
**D** ← dynamic rate at beginning of movement

**Output:**
**Ro** ← 3D rotation vector for body part at current frame

**Begin**
**1** Ro ← Ri
**2** e ← ease value calculated by Equation 4.1
**3** d ← the full distance the body part should move by the end of the movement, based on $Ri_x$ and D
**4** $Ro_x$ ← new x-rotation value calculated by Equation 4.2

**End** of begin

Figure 4.3: The pseudo code of the downHead primitive function using the Ease-In-Ease-Out function.

The second primitive example uses sine functions to create continuous movement. This provides motion that is smooth and curve-like, unlike straight-line interpolation, as well as more realistic looking than the motion provided by Equation 4.1. This type of motion suits our primitives because the movements they emulate often move in curved patterns in reality. The position and orientation equations are set up as follows:

$$pos = pos + dist * \sin(c * frame) \tag{4.3}$$

The variable *dist* defines the maximum distance that the body part will move from the origin or original position. The variable *c* is generally filled in as $\pi / total_time$ and controls the speed of a movement, where *total_time* is the number of frames between the starting frame and ending frame of the movement's time interval. This interval is based on the amount of time between beats in the music. A song with a fast tempo results in a small time interval because there are a small number of frames between beats. The same concept applies to tempos that are slow. The larger this interval is, the slower the movement is performed, while a smaller interval results in a faster movement because it

has less time to accomplish its motion. An example of a primitive that uses this type of movement is the Jump movement. The term $c$ from Equation 4.3 of this movement is different from the average sine function because it needs to use the sine curve for a period and a half instead of half of a period. The movement starts at $\pi$ and continues until $4\pi$, with this section of the curve perfectly describing the Jump motion of bending at the knees, jumping up and bending at the knees to land. This action is better displayed by the graph in Figure 4.4. Results demonstrating this primitive movement can be seen in Figure 4.5.

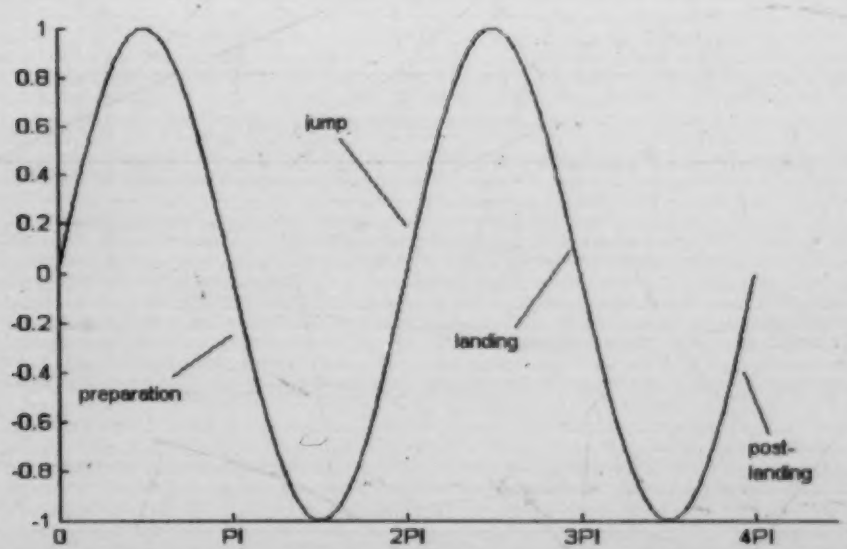



Figure 4.4: The graph above denotes a sine curve from 0 to $4\pi$. The blue line symbolizes the whole sine curve while the red line represents the segment of the curve that best describe the Jump motion. The four changes of direction in the red curve are obvious and they are used to portray the preparation, jumping, landing and post-landing motions of the Jump primitive.

frame 1

frame 3

frame 6

frame 7

frame 9

frame 12

frame 15

Figure 4.5: Results from the "Jump" primitive movement. The character bends his knees to prepare for takeover, jumps into the air, and bends his knees to brace for impact upon landing. This motion follows the sine curve shown in Figure 4.4.

The design of the Jump primitive is described in Figure 4.6. In the cases where the character is preparing to jump or landing from the jump, the distance that the body part will move is smaller than that of the jump segment itself. This occurs because we only want subtle movements at these points in time to depict jump preparation and the resulting impact.

Unlike the other movements, physics equations are used to implement the Throw primitive in order to create a fully realistic model. The flight velocity is computed using the equation

$$v = \sqrt{\frac{height \cdot 2g}{\sin^2 \theta}} \tag{4.4}$$

where $g$ represents gravity, *height* represents the distance the object will travel in the air and $\theta$ is 90° because the object is travelling directly up along the vertical axis. This velocity value is used to compute the time the flight will take. The corresponding equation is

$$T = \frac{2 \cdot v \cdot \sin \theta}{g}. \tag{4.5}$$

**"Jump" Primitive Movement**

**Input:**
**f** ← current frame
**Ti** ← 3D translation vector for the body part at the beginning of the movement
**ts** ← starting frame of current beat interval
**te** ← ending frame of current beat interval

**Output:**
**To** ← 3D translation vector for the body part at the current frame

**Begin**
**1** To ← Ti
**2** tt ← time difference between ts and te that constitutes the time interval for the motion
**3** tc ← time difference between f and ts, used to denote the current frame within the movement, rather than within the animation
**4** d ← value that is 1/3 of tt, used to divide the movement into 3 sections: preparation, jump and landing
**5 If** $tc \le d$ or $tc > (tt - d)$, **then** d ← small value depicting distance knees will bend
**Else** d ← large value depicting height of jump
**6** $To_y$ ← y-translation value of body part at current frame calculated by Equation 4.3
**End** of begin

Figure 4.6: The pseudo code of the Jump primitive movement based on a sine equation.

T is then divided by the movement's total time interval in order to calculate a ratio of the difference between how much time is available and how much time the movement takes. Since we want the movement to take the allotted amount of frames available rather than the time determined by the flight velocity, a new flight velocity is calculated based on this ratio and by using the equation

$$ratio = T / timeInterval$$
$$v_y = \sqrt{2 \cdot height \cdot ratio^2 \cdot g}\,. \qquad (4.6)$$

To make the movement take $k$ times longer, where $k$ = *timeInterval*, the acceleration (gravity) should be proportional to $1/k^2$ because its units are in m/s$^2$. This is why the gravity value is multiplied by $ratio^2$. The height of the ball at a particular time is computed as follows:

$$pos_y = pos_y + \left( v_y \cdot t - \frac{ratio^2 \cdot g \cdot t^2}{2} \right) \qquad (4.7)$$

Smooth transitions between movements are an important aspect of animation. Due to the nature of the primitives and the ability of dynamics to change the starting and ending positions of the movements, it is necessary to include a function that is able to interpolate between movements over an interval of time. This function looks ahead in time to the next beat in the music and retrieves the movements that will be occurring for each body part. The dynamic rate for the next beat is also computed and subsequently used by each look-ahead primitive.

The starting position of each movement for the next beat is determined based on the position of the current beat and the look-ahead dynamic rate. Straight-line interpolation is performed between the ending position of the last beat and the starting position of the next beat. In the case where a body part is being used for both the strong beat and the weak beat, interpolation is not performed between movements because there are no free frames between beats. The next movement will automatically jump to its starting position rather than smoothly moving there. This creates jerkiness in the motion due to the lack of an even transition. If there are free frames between the ending time of one movement and the starting time of the next then interpolation will be performed over the entire set of frames.

## 4.5 Problems with the Hip-Hop System

The major problem with the Hip-Hop system is the inability to properly judge whether the final result is correct. Evaluation of dancing is generally subjective, but most dances have a structure that is followed in order to classify it as a certain dance type. Hip-hop dancing has no structure and is therefore judged entirely according to a viewer's opinion. This lack of structure makes it extremely difficult to perform a fair evaluation of the work because there are no rules by which we can judge the animation. Comparisons between real examples and animated ones are nearly impossible due to the extreme versatility of this dance type.

The range of movements that can be performed in a hip-hop dance is immense, so it is difficult to pick out primitive movements that can be reused to create more complex motion. This particular system is in need of a wider variety of movements that can be used in combination with each other. We concentrated on implementing movements that

can be used by several body parts, but the animation would be more interesting if more movements specific to certain body parts were included. The primitive movements need to be more exciting and expressive of the music. The resulting animation is generally unappealing to the viewer because the primitive movements cannot be combined well enough to create complex motion. Figure 4.7 displays a good example of a movement combination that is uninteresting to the viewer due to its overall simplicity and its lack of musical expression. It uses the "Lift" primitive on the right arm and leg, displaying the reusability of the movement, but it also demonstrates the necessity for more eye-catching motion.

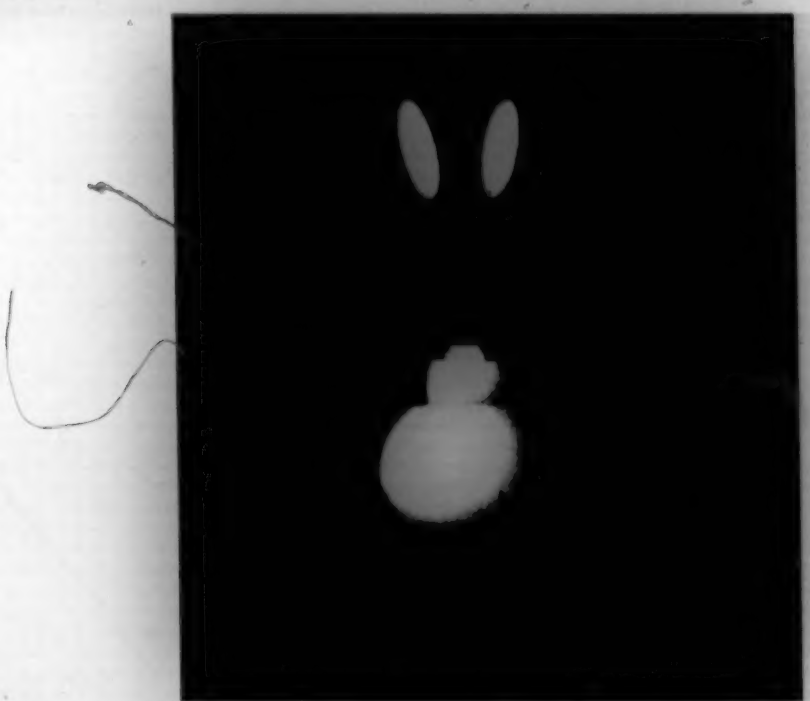


Figure 4.7: An example of an unappealing movement combination where the right foot and arm are simply lifting and lowering on each beat.

Another difficulty with the system stems from the script file. Although effective, the setup of the script file still makes it difficult for the user to design the animation according to a predetermined plan. The user can change movements within intervals, but creating a continuously changing sequence of movements is extremely time-consuming. It is also not practical for integrating multiple characters into the scene because the

characters are not handled individually. Movements for all characters are specified in the same script file, which results in a file where it is difficult to determine which movements correspond to which characters. The script file setup also does not give the impression of creating a dance. It is difficult to picture what the final result will look like based on the script file because it is split into mappings rather than dance movements. The user cannot instantly determine when a movement will be performed, which makes it extremely difficult to plan out and execute an already existing dance.

These problems are addressed with the introduction of a new dance system that is built from a structured form of Celtic dance. The underlying purpose of the creation of this new system is to make dances that are more appealing and interesting for the viewer than those that could be created by the Hip-hop system. The next chapter discusses the new system and its advantages.

# Chapter 5

# Celtic System

One of the major problems of the Hip Hop system is the inability to compare the results of the system with real life dance examples. Hip Hop exists in an extremely versatile and creative environment where dance rules are not present and therefore evaluation of dances is entirely subjective. We want a system where the results can be compared to existing techniques and dance sequences and where the correctness of the results can be more easily evaluated. We are looking to generate an animation that looks like Celtic dancing, but is a unique variation of existing performances. Celtic dancing was chosen because it is an interesting and exciting dance where the movements are performed almost entirely by the legs. Using only three major body parts (two legs and the torso) simplifies the system and allows us to concentrate on the main movements. The system is provided with knowledge of Celtic dancing, including several preprogrammed primitive movements and routines.

In Celtic dancing the position of the feet is extremely important in determining the next movement in a sequence. In general, both feet are angled away from each other at the heels and one foot is placed in front of the other foot. The front foot normally initiates the movements and often determines in which direction the motion will occur. Constraints are placed in the system to allow for this aspect of Celtic dancing. These constraints are used to increase the reliability of the system by ensuring certain movements are only performed by certain body parts, as determined by knowledge of the Celtic dance.

The Celtic system is implemented in such a way so that the dance moves and constraints are separate from the actual animation setup itself. This allows for other dances to be included in the system without much adjustment on the part of the animation system. The animator simply needs to implement a node that includes primitive movements and knowledge of the new dance and add it to the animation system. The addition of several new dances would allow the user to see how different dances work with different types of music. The user could also mix and match movements from the

different dances to create a unique dance type. The setup of the Celtic system encourages dance addition by the animator and experimentation by the user.

This chapter presents the important components of the Celtic system, including primitive movements and how to combine them to create larger dance movements or routines. Movement constraints and mappings between movements and musical attributes are addressed. The two script files used by the system to create animation sequences are discussed in detail. Lastly, applications of the system are outlined.

## 5.1 System Overview

Like the Hip-Hop system, the Celtic system is comprised of a music analysis component and a motion synthesis component. However, the motion synthesis component of the Celtic system is fairly different from that of the Hip-Hop system. It involves an improved script file set-up, animator-chosen mappings between musical attributes and movements, primitive Celtic movements and Celtic-based dance routines.

The script file allows the user to input a motion sequence based on built-in Celtic movements. Its main purpose is to give maximum control over the final animation to the user rather than the system. The new version of the script file is set up so that it is easier for the user to input a pre-designed dance, as well as less confusing when multiple characters are used in the scene. Two script files are employed for these purposes: the main script file defines the body parts of each character in the scene, and the secondary script file includes the Celtic movements to be used. The movements that the user can specify in the secondary script file include both primitives and routines.

Mappings between movements and musical attributes are used to tailor the final result so that it faithfully represents the input music. The timing of the musical attributes directly influences the timing of the movements, as is the case in real life. Mappings in this system occur between the beats and the movements, as well as between the dynamic levels and the height or distance of some primitive movements. The animator has pre-determined these mappings and the user's only choice is with respect to whether the dynamics mapping is used or not. Mappings can also be defined between body parts and primitive movements to determine which body part will perform a particular motion. The

user can choose some of these mappings, although not all primitive movements are free to be utilized by any body part.

In Celtic dance there exist various popular dance routines. Each routine involves a selection of dance steps that are performed at specific times with respect to each other. The user can call a built-in routine of primitive movements rather than the primitive movements individually and get the same, if not a better, result with less work. The timing of the routines has been pre-determined by the animator and included in the system so that the user does not have to spend time figuring it out. The user is also free to build her own routines from primitive movements and other built-in routines. This option makes re-use of movement combinations within the secondary script file extremely easy.

Primitive movements are fairly simple built-in movements that describe small Celtic steps. We believe that complex motion is created from the grouping of several primitive movements into a larger sequence, and this section of the system is built to demonstrate this idea. The rest of this chapter will discuss more details on motion synthesis, including the components of the script file, mappings, routines and primitive movements mentioned above.

## 5.2 Script Files

Script files are utilized to give the user control over what occurs in the animation. They are simple text files that list Celtic primitives and routines that the user wants performed in the resulting animation. The system reads the script file using a specially designed parser and records each movement in the system as it is read in the script file. The script file is an easy and user-friendly method of allowing the user to create her own animation through a combination of built-in primitives, built-in routines and user-designed routines. The script file is also designed to allow for multiple characters in a scene. There is no limit to the number of characters that can be specified by the user. The system is designed so that multiple characters can use the same script file to perform the same sequence of movements or they can use different script files to perform different animation sequences.

In the Celtic system there are two script files that are used to define the animation. The first script file is the master script and it defines the characters and which secondary script file each one uses. The secondary script file is used to define the animation by listing the movements in the order they should be performed.

### 5.2.1 Master Script File

The system has been implemented so that the user is able to interchange the characters used in animations, as well as to use multiple characters at the same time. We want the user to have complete freedom from naming conventions when designing the character, so a master script file is utilized to define each character being manipulated by the system. The master script file also allows the user to define the secondary script file that each character will use.

The system uses three main body parts: left leg, right leg and upper body (torso). In order to manipulate a character, the system needs to be able to choose those body parts from the scene. At the beginning of the main script file the user needs to define the name of the object in the scene that corresponds to each of the main body parts in the system. An example of this is

LEFTLEG: leftLegCtrl

where *leftLegCtrl* denotes the name of the character's left leg in the scene. It is the user's responsibility to ensure that she is mapping the correct scene object to its corresponding system body part. The object will be picked out of the scene and connected to the system so they can share information.

Dynamics are one of the musical attributes that are mapped to movements. The mapping of this attribute can be turned on and off through the master script file. This gives the user the choice to allow dynamics to alter the movements or to use a constant dynamic range through the animation. A complete example of a character definition in the main script file is as follows:

```
CHARACTER 1
  LEFTLEG: LeftLegCtrl,
  RIGHTLEG: RightLegCtrl,
  UPPERBODY: upperBodyGroup,
```

DYNAMICS: off;

Mapping a character to a secondary script file is extremely easy. A full example of an entire master script file can be found in Figure 5.1. Each character is defined by a number, which ranges from CHARACTER 1 to CHARACTER *n*. Under the mapping section of the master script file the name of the secondary script file that corresponds to each character is specified. In Figure 5.1, the secondary script files are "warriorsLeft_1" and "warriorsRight_1". The .txt ending of the file is omitted by the user and added in by the system when reading the file. User designed routines are also defined under the mapping section. The user chooses the name for a routine and then specifies the text file its movements are found in. The routine names in Figure 5.1 are SHUFFLECLICK and DUALCUT, and their corresponding text files are shuffleclick and dualCut respectively. The user utilizes this name when specifying the routine in the secondary script files and the system will automatically read the movements from the file.

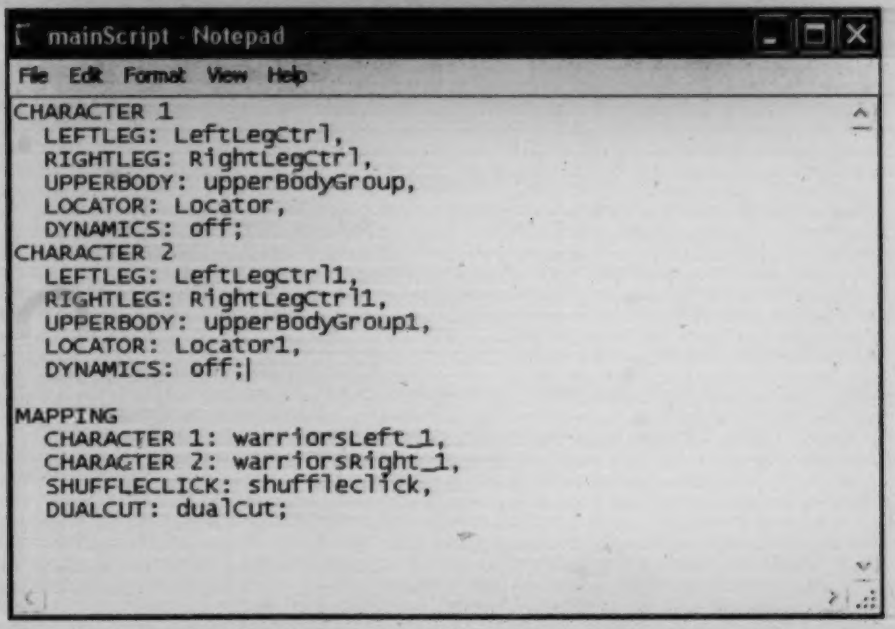



Figure 5.1: An example of a master script file using two characters. The names of the objects in the scene corresponding to the system's main body parts are specified under the CHARACTER headings. The mappings of each character to a secondary script file and each user designed routine name (SHUFFLECLICK and DUALCUT) to its corresponding text file are defined under the MAPPING heading.

### 5.2.2 Secondary Script File

The secondary script file provides a blueprint of how the animation will look. Each movement in the animation is specified in a secondary script file and read into the system in the order listed. This script file defines the dance by using primitive movements, built-in routines and user-designed routines. Unlike the master script file, the secondary script file is not dependent on the characters in the scene. It is built to be easily changed and reused in other animations without regard to who is using it. The purpose of the secondary script file is to define the animation.

The user can construct her motion sequence simply by listing the movements she wants included in her final animation. Each primitive movement and routine has a corresponding name that needs to be specified in order to execute the motion. The user must stick to these naming conventions when creating the secondary script file or the correct movement will not be called. Primitive movements are specified by name and some of them need a body part to be included. For example, when calling the STAMP function, it needs to know which body part it is being applied to. The STAMP primitive lifts and lowers a leg in a movement depicting a leg stamping on the ground, so it is important for the system to known which leg is performing the motion.. Not all primitive movements allow the user to choose the body part it is applied to, but the format for those that support this option is STAMP: RightLeg. The colon after the movement's name is especially important because that is how the parser recognizes the movement as being applied to a specific body part. Routines are specified only by name, such as SHUFFLEHOPBACK, and like the primitive movements, must be spelled with capital letters.

In some cases it may be necessary to start the motion sequence at a certain frame or divide the sequence up into large intervals of time. The system is implemented so that the user can choose a start time for each segment using the keyword *START*. When a user adds this keyword into the secondary script file, the system will perform the first movement at the frame number specified directly after START. The start keyword can be used to divide the script file into different intervals of time. If a user wants a large pause between movements, she can specify one group of movements to start at time X

and another group of movements to start at time Y, where $X \leq Y$. This gives the user control over the timing of the entire animation.

A movement or routine can be performed several times in a row by specifying the name of the movement and then the number of times it should be performed directly after it in the script line. There is no limit to the number of times a movement can be looped through. Along with animation timing, the system provides the ability to control the timing of individual movements. Movement timing can be influenced by the user through the application of *brackets* and *rests*.

Brackets are used to indicate that more than one movement is performed at the same time. In many of the built-in routines, several movements are performed at the same time to create a realistic motion. The user can copy this by putting brackets around the movements occurring in the same time interval. The first movement of the interval is specified normally and the remaining movements are placed within brackets. The system allows for up to 20 movements to be performed during the same time interval in the animation, however, it does not ensure that the movements do not conflict. An example of a routine using brackets is as follows:

LIFTLEG: RightLeg;
(HOP);
(LIFTLEG: LeftLeg);

Rests are used to stagger the starting and stopping times for movements. The concept is adopted from music, where rests denote breaks between musical tones. The rest is specified in the system by the '^' character. Each rest is worth 1/8 of a beat, which means that the length of a rest will change from one piece of music to the next. The faster the song is, the shorter a rest will be. Rests can be placed before or after a movement name and several rests can be used by one movement. If the rest is placed before a movement, the movement will wait 1/8 of a beat before beginning. If the rest is placed behind a movement, it will end 1/8 of a beat earlier. The number of rests used will affect the length of the movement. The system counts the number of rests before and after each movement and determines the starting and ending time of the movement from the separate totals. Examples include:

^HOP;

CROSS^;

^^STAMP^;

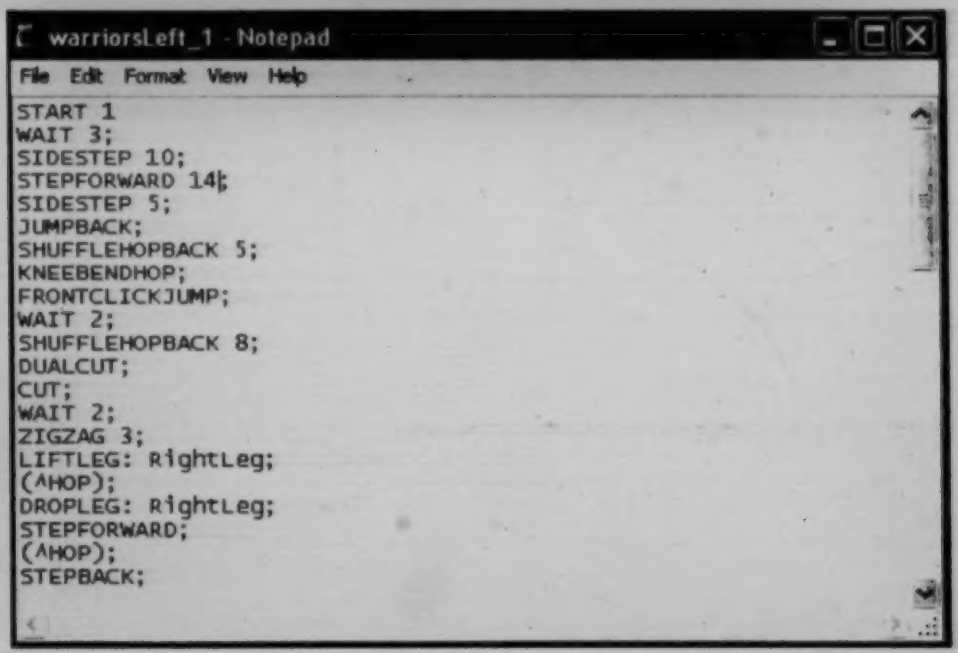



Figure 5.2: A secondary script file allows the user to design her motion sequence by specifying primitive movements, built-in routines and user designed routines in the order she wants them performed in the animation. A user can utilize loops, parentheses and rests in order to retain maximum control over the timing of the animation.

An example of a secondary script file is found in Figure 5.2. The 1st line denotes the starting frame for the animation is frame 1. The 2nd line specifies that the character will wait for 3 half-beats, since each primitive movement is performed over an interval with the length of half a beat. When a number follows a primitive or routine name, the system will perform the movement as many times as specified by that number. The 3rd line states that the Sidestep routine will be performed 10 consecutive times. This results in the character moving across the stage. Each routine takes a different amount of time, depending on how many primitive movements are involved. The Sidestep routine uses two primitives, so each performance of the routine will take 1 beat. The 3rd line needs a total of 10 beats to complete the motion. The 4th line concerns the Stepforward primitive, which involves the character taking a single step forward with her back leg. The script

file specifies that it will be performed 14 times, and since it involves only one primitive, it will take 7 beats to complete. Line 5 uses the same routine as mentioned earlier in conjunction with line 3. The 6th line is performed only once because there is no number following the movement definition. The Jumpback routine uses two primitives and therefore will take 1 beat of time. The Shufflehopback routine noted in line 7 is different from earlier routines because it involves 4 primitives, but two of those primitives occur at the same time. This results in a routine that takes 3 half-beats, or 1.5 beats to complete. The user has specified that the character will perform this routine 5 times, so the entire loop will use 7.5 beats. The Kneebendhop routine also takes 3 half-beats, or 1.5 beats, so the 8th line will take 1.5 beats to complete because the routine is only being performed once. Unlike any other routine, the FrontClickJump motion specified in line 9 uses 4 primitives and takes 2 full beats to finish performing. The rest of the lines specify either routines or primitives and the number of times they should be performed in succession.

There are two exceptions in the above script file that need further explanation. The first occurs in line 12, where the DualCut routine is specified. This routine is a user-defined routine and its length in beat-halves depends on the number of primitives the user specified in it. This particular routine calls two built-in routines: Cut and CutBack. Both routines take 1 beat each to complete, so the DualCut routine uses 2 beats in total. The timing of a user-defined routine is dependent on the user who designed it, so it is the responsibility of the user to remember how long the routine takes to complete. Lines 16 and 17 provide an example of the second exception in the script file. Line 17 uses brackets and one rest, linking it to line 16. In line 16 the character is commanded to lift her right leg, while line 17 specifies that she is hopping at the same time. The Hop primitive is performed at the same time due to the brackets around it, although the rest symbol denotes that the primitive will wait ⅛ of a beat before starting. This results in the character starting to lift her right leg and then ⅛ of a beat later, starting to hop.

One of the unique features of the Celtic system is the ability to use multiple characters with different personalities. In some cases, the user may want multiple characters to perform the same dance move at the same time, and in other cases, the user may want the characters to perform different dance moves during the same time interval. In order to synchronize multiple characters, it is imperative that the user keeps track of

the movement timing for each character. Because the length of time for each movement differs, the user must use combinations of movements that add up to the same total amount of time across characters. An example can be seen in Figures 5.3 and 5.4, where the characters both perform the same movements and different movements over time. Figure 5.3 demonstrates using movements with the exact same timing sequence across characters and the Figure 5.4 exhibits how one routine's timing can be offset by a combination of routines and primitives for continuous synchronization through different moves. In order to keep the characters synchronized in time, the user must continuously track the number of beats used by the movements. Movements cannot be performed at the same time if one character has performed one less beat's worth of movements than another character. Tracking the time used across characters allows for easier synchronization and off-synchronization of movements throughout the dance.

| CHARACTER 1 | CHARACTER 2 |
|---|---|
| JUMPBACK; | JUMPBACK; |
| LIFTLEG: RightLeg; | LIFTLEG: LeftLeg; |
| HOP; | DROPLEG: LeftLeg; |
| DROPLEG: RightLeg; | HOP; |
| HOP; | WAIT; |
| STEPFORWARD 4; | STEPFORWARD 4; |

Figure 5.3: An example of synchronizing a dance over multiple characters. Each movement in a sequence takes the same amount of time for completion as the movement directly across from it in the other character's sequence. The sequences involve movements that are different from the other character as well as the same. This is the easiest way to synchronize over the animation.

The system does not restrict the movement order decided by the user. It does not check to make sure that two movements can be performed properly at the same time or that consecutive movements make a realistic sequence. The user must ensure that her movement choices are realistic and that the order makes sense. For example, if the user wants the character to lift her leg and then perform a tapout movement on the ground, she must ensure that the character also lowers her leg. This type of movement transition is not performed by the system because it is too difficult to decipher all possible transition combinations between all the primitive movements and routines. Celtic movements are

| CHARACTER 1 | CHARACTER 2 |
|---|---|
| HOP;<br>SHUFFLEHOPBACK;<br>JUMPBACK; | HOP;<br>STEPFORWARD;<br>(^HOP);<br>STEPBACK;<br>(^HOP);<br>STAMP: RightLeg;<br>JUMPBACK; |

Figure 5.4: Another example of synchronizing data over multiple characters. Unlike the figure above, the movements in this sequence are performed off-synch. The Shufflehopback routine performed by Character 1 takes 1.5 beats of time and the lines under Character 2 from Stepforward to Stamp also take 1.5 beats of time. The Jumpback primitive at the end of both sequences is performed at the same time because the movements prior to it are synchronized in time, if not in movement.

provided that easily create transitions and so it is the user's responsibility to check her animation for inconsistencies.

### 5.2.3 Parsing Script Files

Once the user has designed the dance and built the script file it is necessary to input the information into the animation system. Special parsers were built for the main script file and the secondary script files. The information obtained from each file is dissimilar so the setup of the parsers is significantly different. The parsers are extremely important components of the animation file because they ensure that the script information is read correctly.

The parser for the main script file is responsible for reading character information, user-defined routine file names and secondary script file mappings. The word CHARACTER denotes that character information follows and the parser uses this keyword to determine the structures it will use for storage in the system. Special keywords are then used to describe the body parts belonging to each character. These keywords include RIGHTLEG, LEFTLEG and UPPERBODY. The parser recognizes these words and stores the name of the corresponding scene object in a special structure that is unique for each character. The parser gathers the character information for single and multiple characters and selects the objects in the scene based on the given body part

names. The parser also identifies the keyword DYNAMICS. The parser records the choice made by the user with respect to incorporating dynamics into movement. The user must use the word "on" or "off" to denote that the dynamics feature is to be on or off, respectively.

Once the user has described all the character information, she uses the keyword MAPPING to denote the end of the character information and the beginning of the movement information. If the parser finds the word CHARACTER after detecting the mapping keyword, it knows that the next piece of information will be the name of the secondary script file that the specified character will use. If this keyword is not the input then the parser knows that a user-defined routine will be specified for use. It stores the name of the routine and the name of the routine's unique script file for later recognition in the secondary script file parser.

The parser for the secondary script file is responsible for interpreting the design of the dance. The parser retrieves the name of the secondary script file from the information obtained from the parser of the main script file. This file is opened and the keyword START is always the first word input from the script. This defines the starting frame for this particular movement section. Once the interval's starting frame has been established, the parser reads in the movements one by one. It is searching for the following grammatical components: '(', ";" and ":". If it finds a bracket then it realizes that the next movement is a submovement, which means that it will be performed at the same time as the previous movement. The movement is stored in the specially built movement structure as a submovement of the previous one. The movement structure is set up similar to a 2D array so that the main movements are stored in the first dimension and the submovements are stored in the second dimension of the corresponding main movement. If the parser finds a semicolon, it simply stores the name of the routine or movement for use in the animation system. Lastly, if the parser finds a colon, it recognizes that a body part will be specified for this particular movement and it stores them separately: the movement name in the movement structure and the body part name in the body part structure. The index into both structures is the same so that it is easy to coordinate movements and body parts.

Two exceptions exist in our parser setup. The first occurs when a user-defined routine is specified in the script file. Each routine name read by the parser is compared to a list of user-defined routine names. If the name does not exist on the list, the routine is read into the movement structure as normal. If the name does exist on the list, the parser calls itself using the script file specified for the user-defined routine. All the movements in this special script file are read into the movement structure in order and the user-defined routine name is discarded. In effect, the movements corresponding to the user-defined routine are replacing the routine itself. The parser then continues on to the next movement in the secondary script file.

The second exception occurs with respect to the built-in Turn routine. This routine needs the direction and number of turns specified by the user in order to execute. The name of the routine and the direction are stored in the same way a movement and corresponding body part are stored. The number of turns being performed is stored in its own structure because the looping amount is handled differently from other routines. The loop number is used to determine how far the character will turn at once, rather than how many full turns it will perform in a row.

Like the parser for the Hip-Hop system, this parser relies on proper grammar and spelling to work properly. The user is responsible for checking that the script file is built according to animator-determined instructions. The user is left to correct her own mistakes because it avoids discrepancies that occur if the parser incorrectly changes an aspect of the script file that was correct. The parsers for both script files are incredibly effective, despite their inability to recognize errors in the script file design, and are a vast improvement over the original script file design found in the Hip-Hop system.

## 5.3 Mappings

The main purpose of our animation system is to use music as the prime vehicle to drive an animation. Musical attributes such as the beat are mapped to Celtic movements and used to alter the motion based on the music. This system does not simply synchronize an already existing animation with a piece of music, but it actually builds the animation according to details extracted from the input song. Unlike synchronization methods, the movements in our system change along with the music. We create a final animation that

is tailored to fit the music chosen by the user while providing an interesting and entertaining sequence of motion.

### 5.3.1 Mapping Beats to Movement Timing

The timing of the movements is based almost entirely on the tempo of the music, where the faster the song, the faster the movements are performed. The position of the beats are inputted into the animation component by the music analysis component and used to determine the length of each movement's time interval. The beat attribute is not mapped to a specific movement or set of movements, but is used to structure the movements so they occur on a beat, as is generally seen in real-life dancing.

Celtic dancing is a fairly high-speed dance, where several movements occur in the space of one beat. In order to stay faithful to this style of dance, the mapping between beats and movements was changed from the original one movement per beat found in the Hip-Hop system. In this system two primitive movements are performed for each one beat. This rule applies to routines with multiple primitives as well. Several routines use three or four primitive movements and result in taking 1.5 or 2 beats to finish. This timing can make coordinating a routine between several characters more difficult because the user must be aware of the beat length of each routine, but it results in a more realistic animation.

Rather than performing each routine in a single beat, we choose to map two primitives to one beat because it provides smoother motion and better transitions between primitives. If a routine such as FrontClickJump was performed in a single beat then four primitives would have to be completed by the end of the beat. In the case where the user inputs a fast song that results in a short time interval for each routine, this particular routine becomes extremely choppy because each primitive only gets a small window of time in which to complete its motion. By restricting the animation timing to two primitives per beat we can be assured of providing smooth motion no matter how high the song's tempo is.

### 5.3.2 Mapping Dynamics to Movement Distances

Dynamics are a source of interest in music because they contribute to making a song interesting and unique. Almost every listener can distinguish the loud sections from the soft ones and the transitions between the levels. Dynamics can also affect the movements used to dance to a particular song. Small and timid motions are not used on a song that is loud, and large extreme motions are not used on a song that is consistently soft. In order to make our animation more appealing we take the most interesting aspects of music, such as the dynamics, and use them to directly change the movements to better reflect the mood of the song.

The dynamics levels in the system range from 1 to 5, where 1 denotes soft dynamics and 5 denotes loud ones. There exist several primitive movements where the dynamics level affects the distance moved by a body part or the height of a jump or kick. The higher the dynamic level is, the higher the height or the longer the distance will be. All primitive movements have an animator-determined height or distance range. The current height or distance value for the movement will always fall within this range, but will be affected by the current dynamics level. For example, in the Hop primitive the height range is from 0.6 to 3.0 units. The height of the primitive is calculated by the following equation:

$$height = 0.6 \cdot getDynamicsLevel() \tag{5.1}$$

The majority of the equations are calculated in this way, where the constant value (0.6 in this case) is the low end of the range and the constant value multiplied by 5 is the high end of the range.

The primitive movements that are altered based on the dynamics include:

- Hop/ShortHop/HopForward
- LiftLeg
- DropLeg/DropLegBehind
- LongStep
- SlideBehindStep
- Cross

There exists an exception with respect to applying dynamics to primitive movements. In most cases, the current dynamics level can simply be applied to the height or distance value of a movement, however the DropLeg, DropLegBehind and SlideBehindStep primitives cannot follow this. In general, once a character lifts a leg she will lower it at some time or another. If the height of the leg is determined based on dynamics and the dynamics level changes by the time she lowers it, she could be lowering the leg too much or too little. It is necessary that the dynamics level for the DropLeg and DropLegBehind primitives is the same as that of the LiftLeg primitive, while the SlideBehindStep movement must have the same dynamics level as its counterpart LongStep.

## 5.4 Constraints

Foot position is an extremely important aspect of Celtic dance. It can help to determine the next movement in a motion sequence or the direction the character moves in around the stage. In many cases, the front foot is used as the starting foot for a routine or movement. This is the main reason that the system keeps track of which foot is in front and which is behind at each frame. We incorporate this Celtic knowledge into the system through the implementation of constraints. These constraints are used in some primitive movements and all built-in routines. Their purpose is to ensure that a primitive or routine is being performed by the correct body part according to the rules of Celtic dance.

Only four primitives incorporate constraints into their implementations. For example, the StepForward primitive movement switches the front foot with the back foot by taking a step forward. Essentially, the back foot moves forward until it is positioned in front of the opposite foot, similar to a walking motion. A constraint is used in this routine to make sure that it is performed by only the back foot. If a user calls this primitive with the front foot, the system will not perform the motion. The SlideBehindStep primitive is also constrained to the back foot, while the StepBack and LongStep movements use only the front foot.

Constraints are especially important to built-in routines because the routines are created based on specific combinations of Celtic primitives. The constraints enforce the rules for performing a specific routine, such as ensuring the front foot performs the first movement. The animator, based on Celtic knowledge, chooses which body part performs

which movement and uses the constraints to guarantee her design is followed. An example of this is the FrontClickJump routine. This routine performs a scissor-kick jump. The process starts with the front leg being lifted into the air. It is then lowered at the same time the back leg is lifted into the air and the character hops in place. The back leg is finally lowered down to the ground to complete the motion sequence. The first lift and drop movements must be performed by the front leg, while the second set is performed by the back leg. Foot constraints ensure that this order is followed and that the motion conforms to Celtic dancing.

The constraints incorporate system knowledge of the positions of the character's feet with Celtic knowledge of how movements and routines should be performed. The use of constraints in a movement or routine is decided entirely by the animator and cannot be altered by the user. Constraints are used to enforce the integrity of Celtic dance and make it easier for the user to put together realistic motion.

## 5.5 Routines

The dance routines implemented in this system are more complex dance steps than those provided by the primitive movements. In many cases, Celtic dance has a dance step that consists of several primitive motions, but it is referred to by the name of the dance step rather than the primitives individually. Combining several primitive movements allows for these complex routines to be created and used by the system. The user can use these routines by specifying them by name. The routine will automatically call the appropriate primitive movements to create the movement. The system handles two different types of routines. The first is the built-in routine, as programmed by the animator, and the second is a user-designed routine.

### 5.5.1 Built-in Routines

The built-in routine is implemented directly into the system by the animator. It makes use of several primitive movements and controls the timing of them to create an actual Celtic dance step. In some cases a movement is slightly altered so it fits into the routine better. These routines are called by name and will call the primitive movements themselves. Routines were implemented for some of the major Celtic dance moves. The

purpose of a routine is to make the animating process easier for the user. Rather than having the user continuously specify small primitive movements in the same order, she can call a routine that does the same thing. They save the user time and frustration because the animator has already worked out the timing of the primitive movements so that the routine is correct. This makes it easier for the user to create an entire Celtic dance based on known Celtic movements. These routines are similar to how a person would learn to Celtic dance and are taken directly from [11]. The majority of these names are the ones used by dancers performing Celtic dance. Some routines, such as CutBack, ClickZigZag, FrontClickJump and KneeBendHop were not given names in the Celtic video, so their names are based on the movements used in the routine or similar routines.

There are eleven Celtic routines implemented into the system. They include:

- **ClickZigZag** – uses the ClickHeelsOut and ClickHeelsIn primitives to continuously rotate the feet in so they click at the heels and then back out to the original orientation. The feet are moved so they are lined up beside each other. This routine takes 1-1.5 beats for each performance. The timing change occurs due to the modification of the position of the feet. Once the routine is finished and the system is moving on to another routine, the feet need to be moved back to their original position before this routine occurred. This movement needs an extra half-beat. Essentially, if the routine is performed 3 times in a row, it will take 3 beats for the routine motion and ½ beat to move the feet back into position, for a total of 3.5 beats.

- **Cut** – uses the CutBend and Hop primitives to perform a specialized Celtic jump using the front leg. Both primitives perform at the same time to achieve the jump before the StampDown primitive is used to place the front foot back into position. As the character hops in the air, the front leg bends at the knee and rotates so that the foot is in position directly in front of the opposite knee. There is a slight pause before the front leg is lowered to the ground. This routine uses 1 beat for each occurrence.

- **CutBack** – uses the CutBend and Hop primitives to perform a jump using the back leg. It is exactly the same as the Cut routine, except it uses the opposite leg. It also uses 1 beat for each occasion it is used.

- **FrontClickJump** – one of the more interesting jumping routines, it uses the Stamp, LiftLeg, Hop and DropLeg primitives. It stamps the back leg before lifting the front leg out in front of the body. The back leg is lifted and the hop movement is used as the front leg is lowered, producing a heel click in mid-jump. The back leg is then lowered to the ground. Both legs are off the ground

and in front of the character for part of the routine. This routine needs 2 full beats of time.

- **JumpBack** – used to switch the front foot and the back foot. This routine uses the LiftLeg, DropLegBehind and HopForward primitives to create a jump that lifts the front leg in front of the character and lowers it so it ends up behind the back leg. The front leg is completely straight as it is lifted and lowered. Because the HopForward primitive occurs at the same time as the DropLegBehind primitive, the JumpBack routine uses only 1 beat.

- **KneeBendHop** – another jumping routine that uses the LiftLeg, KneeBend, Hop and StampDown primitives. The front leg is lifted into the air in front of the character, after which it is bent at the knee as the foot and the leg move directly in front of the body during the hop movement. The knee is facing upwards. The leg is then placed back on the ground in front of the back leg. This routine takes 1.5 beats to finish.

- **Shufflehopback** – An extremely popular Celtic dance move, this routine involves switching the front and back feet in an interesting way. It makes use of the TapOut, TapBack, ShortHop, and StampDown primitives. The character taps out her front foot and then taps it back and behind the back foot while hopping on the back foot. The original front foot is then placed down behind the original back foot into position. The Shufflehopback routine needs 1.5 beats for each occasion it is used.

- **SideStep** – used as a method for moving the character in space, this routine allows the character to move from side to side. The front leg takes a short step to its closest side and then the back leg follows it to the same side. This routine uses the Cross primitive twice and needs 1 beat for each performance.

- **SlidingStep** – another method that moves a character in space, this routine performs a diagonal step forward. The front foot takes an extended step forward and slightly to the side using the LongStep primitive and the back foot slides behind it using the SlideBehindStep primitive. This routine uses two primitives in succession, so it needs 1 beat of time to finish the motion.

- **Turn** – this routine allows the character to change her overall orientation. She can rotate in increments of 90° to face different directions. This is the most complicated built-in routine, as its implementation is different from the other routines and the turn can occur either to the left or to the right. It does not use any primitives, but is classified as a routine because it involves several movements. The character turns around the front leg. She starts by lifting the leg that is not turning, rotating the other leg so it is on the ball of the foot, and turns the torso slightly in the direction that she will be turning towards. The next movement involves rotating the rest of the body some multiple of 90°, as determined by the user. Lastly, the character rotates her torso back slightly to

compensate for the original rotation in the first movement, rotates the turning foot so that it is completely on the ground rather than on the ball of the foot, and places the back leg down on the ground. As the character rotates around the front foot, the back leg and upper body also rotate around the front foot so that the character is constantly in the correct Celtic position. The correct Celtic position involves both feet angled away from each other at the heels and one foot placed in front of the other foot. The Turn routine requires 1.5 beats because the character lifts her back leg, turns on the front leg, and lowers the back leg, where each movement takes ½ a beat.

- **ZigZag** – involves the SwingHeelsIn and SwingHeelsOut primitives to constantly rotate the heels in towards each other and then swing them back out to their original position. The upper body moves up and down slightly during this routine because the height of the heels in the air changes as the character moves from the heels swinging out (heels at highest point) to heels swinging in (heels at lowest point). Similar to the ClickZigZag routine, this routine uses two primitives. However, this routine only needs 1 beat for each occurrence because it does not change the position of the feet.

The timing aspect of a routine is extremely important. The built-in routines give the animator the ability to have several primitives performed at the same time and at different offsets. Each routine can use a different number of primitive movements to accomplish its purpose. An example of a routine that uses different primitives and different offsets is the Shufflehopback routine. Figure 5.5 displays the pseudo code for this routine, including the primitive movements involved in creating the complex motion. The Shufflehopback is comprised of three main time intervals, denoted by the term *movementCnt* in Figure 5.5. Each interval takes place over half a beat interval in the music, as chosen by the animator for primitive timing. The Shufflehopback routine will take 1.5 beats to complete, with the TapOut primitive taking place over the first time interval, TapBack and ShortHop occurring during the second time interval and StampDown occurring over the third time interval. The ShortHop primitive does not share the same movement start time as the TapBack primitive, despite both occurring in the same time interval. The ShortHop has an offset of 2 from the TapBack so it will start later and take less time to complete.

**"Shufflehopback" Routine**

**Input:**
**f** ← current frame number
**Vi** ← 3D rotation or translation vector for the body part at the beginning of the movement
**ts** ← starting frame of current beat interval
**te** ← ending frame of current beat interval

**Output:**
**Vo** ← 3D rotation or translation vector

**Begin**
1 Vo ← Vi
2 **If** body part is the front foot
3 **If** movementCnt = = 1
Vo ← ***TapOut(f, Vi, ts, te)***
4 **Else if** movementCnt = = 2
Vo ← ***TapBack(f, Vi, ts, te)***
5 **Else if** movementCnt = = 3
Vo ← ***StampDown(f, Vo, ts, te)***
6 **If** f > ts+2 and movementCnt = = 2
Vo ← ***ShortHop(f, Vo, ts+2,te)***
**End** of begin

Figure 5.5: The pseudo code for the built-in Shufflehopback routine. It uses the TapOut, TapBack, ShortHop and StampDown primitive movements.

### 5.5.2 User Designed Routines

In some cases the user may want to use routines that are not implemented in the Celtic system. The system allows for user-designed routines in which the user can define her own routines through text files. The user can create her own dance moves by specifying primitives or built-in routines and their order. There is no maximum length limit for a routine, so the user is free to use as many primitives as necessary. The user-designed routine makes it easier to create an animation sequence because the user can define routines with combinations of movements that are used continuously in the animation. For example, if the user finds that she is constantly using three primitives in the same order in several places in her animation, she can put them into a routine. Rather than

specifying the three primitives each time she wants that specific combination, she can specify her specially designed routine instead. The system will retrieve the routine as input and perform the primitives found in that routine.

In many routines the timing of movements is extremely important. While the animator controls the timing for built-in routines during implementation, it is more difficult to do this for user-designed routines. The system originally performed the movements in the order they were specified, one after the other. The brackets and rest features detailed in Section 5.2.2 were implemented into the system to give the user the same control that the animator enjoys.

Rests and brackets can be used in combination in a routine that performs several movements using a specific arrangement. In fact, the user is able to reconstruct the built-in routines using a combination of primitives, rests and brackets. For example, the Shufflehopback routine can be reconstructed as follows:

Shufflehopback: TAPOUT: RightLeg;
TAPBACK: RightLeg;
(^SHORTHOP);
STAMPDOWN: RightLeg;

Variations of built-in routines can also be created, as user designed routines are not limited to exclusively using primitive movements. User designed routines promote creativity and experimentation by the user because she is free to use any primitive or built-in routine in the system, as well as control their order and timing. Once the routine files are designed they can be reused in any animation and changed easily by the user.

## 5.6 Primitive Movements

The primitive movements implemented in this system were determined by studying videos of Celtic dancing and establishing the simple movements that make up the larger routines. An especially helpful video was Colin Dunne's Irish dance instructional video, "Celtic Feet." [11] This video breaks down routines into primitive steps and displays how they can be put together to create dance sequences. Most of the primitive movements implemented in our Celtic system are modeled after the movements depicted in this video. A total of twenty-four primitive movements have been implemented into the

system. They can be used in different combinations to create routines, which are slightly larger and more complex movements. The primitive movements and their purpose are listed as follows:

- **ClickHeelsIn** – the feet line up horizontally, lift up onto the balls of the feet and turn in so that the heels click against each other.

- **ClickHeelsOut** – once the heels have clicked against each other, this movement rotates the feet so they are once again apart.

- **Cross** – used for moving the body in a sidestep movement, this primitive moves the body to the side of the front leg. For example, if the front foot is the right foot, the body will take a step to the right.

- **CutBend** – takes the front foot and bends it so that the foot is in front of the back foot's knee, both in distance and height. The knee should be sticking out to the opposite side of the back foot.

- **DropLeg** – lowers a leg from the position where the leg is stretched out in front of the body. The leg is lowered directly to the ground.

- **DropLegBehind** – lowers a leg from the position where the leg is stretched out in front of the body, but places the leg behind the other leg.

- **HeelsUp** – rotates the foot so that the character is on the balls of her feet.

- **HeelsDown** – rotates the foot so the character goes down from the balls of her feet to the heels of her feet (the whole foot is solidly on the ground).

- **Hop** – performs a jumping motion on the spot.

- **HopForward** – performs a jumping motion that moves the character slightly forward.

- **KneeBend** – lifts the leg and bends the knee so that leg is fairly close to the body, with the knee bent and pointing up.

- **LiftLeg** – lifts the leg out and straight in front of the character.

- **LongStep** – takes a long step that equals the distance of several normal steps, generally used with front foot.

- **ShortHop** – performs a jumping motion on the spot that does not go as high as the normal Hop primitive.

- **SlideBehindStep** – used in conjunction with the LongStep primitive, it slides the back foot in behind the front foot.

- **Stamp** – lifts a leg up slightly, bending at the knee, and lowers it back down to the ground.

- **StampDown** – lowers a leg to the ground from a slightly raised position.

- **StepForward** – takes a single step forward. This primitive is always performed by the back foot so it moves in front of the front foot.

- **StepBack** – takes a single step backwards. This primitive is always performed by the front foot so that it moves behind the back foot.

- **SwingHeelsIn** – rotates the heels of the feet in towards each other without contact. The feet stay at their original position for the duration of this movement.

- **SwingHeelsOut** – used in conjunction with the SwingHeelsIn primitive, it rotates the heels back out to their original orientation from the inward orientation.

- **TapOut** – moves foot slightly forward along the ground. This primitive taps the ground as it moves forward in a convex type motion and is generally used on the front foot.

- **TapBack** – moves the foot back behind the other leg as it taps the ground in a convex type motion. It is used in conjunction with TapOut, and so assumes the foot is already slightly out in front of the body.

- **Wait** – pauses the motion for a time length determined by the animator. The character does not move during this primitive.

Movement in general is very smooth and occurs in curves rather than in straight lines. For this reason, Equation 4.3 is used to compute the position and rotation of each body part at a given point in time. A simple example of the implementation of such a primitive is the Stamp movement, shown in Figure 5.6, where a foot lifts off the ground to a certain height and then stamps back down on the ground in the same time interval.

In the majority of the primitive movements there are constraints placed on the motion for each body part. These constraints include the direction (right or left) that the primitive moves the body to, which body parts are moved by a primitive, and how much each body part is moved. For example, in many cases the torso of the character will

**"Stamp" Primitive Movement**

**Input:**
**f** ← current frame number
**Vi** ← 3D translation vector for the body part at the beginning of the movement
**ts** ← starting frame of current beat interval
**te** ← ending frame of current beat interval

**Output:**
**Vo** ← 3D translation vector

**Begin**
1 Vo ← Vi
2 tt ← te – ts
3 time ← f – ts
4 dist ← height that foot will raise off the ground
5 $Vo_y$ ← new y-position value calculated by Equation 4.3
**End** of begin

Figure 5.6: The pseudocode for the Stamp primitive movement, using the sine equation for smooth movement.

perform a motion that moves only half the distance of the feet, keeping the balance of the body even. There also exist some primitives in which the movement is performed only by the front foot. The implementation of these primitives is more complicated than the one displayed above because they need extra information about the character and its set-up. An example of one of these primitives is displayed in Figure 5.7.

In this particular function, the distance moved by the feet differs from the distance moved by the torso/upper body. This is because the upper body is always halfway between the two feet, creating a balance for the character. By comparing the name of the current body part being moved with the string "upperbody," the function can determine if it is moving a foot or the torso and act appropriately. The Cross function moves the character in a sidestep motion across the stage. It is necessary to know the direction that the character is going to move, so the name of the current front foot is compared to the strings "right" and "left". The direction is determined solely by the front foot, which results in the character stepping to the right if the right foot is the front foot and to the left otherwise.

**"Cross" Primitive Movement**

**Input:**
**f** ← current frame number
**Vi** ← 3D translation vector for the body part at the beginning of the movement
**ts** ← starting frame of current beat interval
**te** ← ending frame of current beat interval

**Output:**
**Vo** ← 3D translation vector

**Begin**
1 Vo ← Vi
2 tt ← te – ts
2 **If** body part is the torso **then** dist ← $(0.6 \cdot getDynamicsLevel())/2.0$
3 **Else** dist ← $0.6 \cdot getDynamicsLevel()$
4 **If** the front foot is the right foot **then** negate dist so the movement direction changes from the left to the right
5 c ← $\pi/(2 \cdot tt)$ (controls the speed for changing the body part's x-position and is used in Equation 4.3)
6 $Vo_x$ ← new x-position calculated by Equation 4.3
7 dist2 ← small value corresponding to distance from ground that foot lifts as it steps
8 c2 ← $\pi / tt$ (controls the speed for changing the body part's y-position and is used in Equation 4.3)
7 $Vo_y$ ← new y-position calculated by Equation 4.3
**End** of begin

Figure 5.7: The pseudo code of the Cross primitive movement. The side of the body that the front foot is on (right or left) determines the direction that the movement travels.

## 5.7 Applications of Celtic System

This animation system is a unique music-driven approach to character animation. Its purpose is to create a unique animation with the structure of a Celtic dance but that is tailored to suit the chosen music. The resulting animation needs to be interesting, exciting and expressive of the corresponding music. Our system provides a user friendly and flexible way to create such an animation. It includes distinctive features that are not included in synchronization-based systems, making it a distinctly more appealing method of generating music-driven animation.

The majority of synchronization methods display their advantages using a single character. While the movement of a single character is interesting, multiple characters can create a performance that is more visually stunning than that of one character. Synchronizing multiple characters to the music is a difficult task for synchronization systems to perform, which is why most researchers do not include it in their work. One of the unique features of our system is the ability and ease to move several characters in a scene to create a performance. This set-up is extremely useful for Celtic dancing where the most interesting dances are performed by a troupe. The user is able to include as many dancers as she wants in an animation, making it easy for her to design a dance for a large troupe of characters.

One of the distinctive aspects of our system is the ability to give each character a different personality. Dancers are not limited to performing the same dance together. A different secondary script file can be created for each dancer in the scene and linked through the main script file. Hence, the user can synchronize movement between characters as well as infuse individuality into the motion. Several combinations of multiple characters are possible, many of them used in Celtic dancing itself. The majority of the troupe dancers can use the same secondary script file while the principal dancer in the troupe can be assigned to a different one to make her stand out. Dancers can be split into groups, where each group dances to its own set of movements, or each dancer can be given her own dance sequence. The master script file is set up to give the user the freedom to design new characters and incorporate them into the animation.

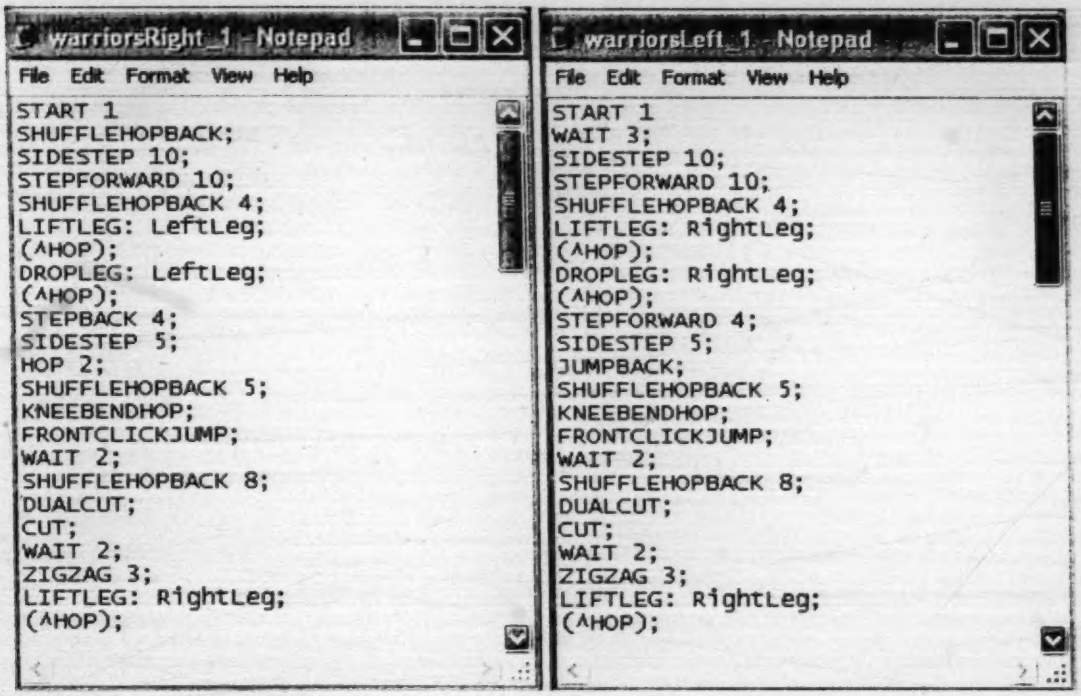



Figure 5.8: The diagrams above are segments from two secondary script files used to control multiple characters.

Figure 5.8 demonstrates how two secondary script files can be used to control multiple characters. They have been synchronized so that the movements are exactly the same in some intervals and different in others. Synchronization occurs when two movements have the same timing and will take the same amount of time to complete. For example, in line 2, the Shufflehopback routine takes 1.5 beats, so the Wait primitive is called 3 consecutive times to make its completion time the same as the Shufflehopback routine. This ensures that the Sidestep routine in line 3 is started at the same time by each character. In order to have synchronization across multiple script files, it is necessary to ensure that the timing for groups of movements is the same. This is best done by determining the amount of time one movement or group of movements takes and filling in the other script files with a set of movements that takes the same amount of time to perform. Figure 5.8 is a good example of how different movements can be synchronized in time. By using the secondary script files in conjunction in the same animation we create two characters that are performing the same dance with different

personalities. As far as we know, this is not attempted by any animation system similar to ours.

Our script file is not only unique in its ability to incorporate multiple characters with different personalities into an animation, but its capability to manipulate movement is an exceptional technique as well. The brackets and rests used by the system to organize motion allow for movements to be combined in the same time interval and at staggered starting and stopping times. This technique permits multiple characters to move out of synch by performing movements in the same time interval but with a time gap between them. The user can also make a character perform more than one movement at the same time by using brackets in the script file. The time staggering technique is extremely difficult to do in synchronization-based approaches because it requires long and tedious work by the user to exactly align starting and ending positions for multiple movements for multiple characters. The majority of synchronization methods use motion capture data, so it is challenging to combine movements into the same interval unless the resulting movement already exists in the data. The ability of our system to provide both of these methods makes it extremely usable and flexible.

Both introductory and experienced animators can use our system to experiment with dance and musical attributes. It provides more flexibility than synchronization methods whose purpose is similar to that of our system and it gives users more control over the final result. Since Maya provides the interface for the animation, it is convenient for the user to build up characters and a scene using Maya's extensive features. Our system supplies the user with an efficient and convenient method for coordinating music with movement, making it a unique resource in character animation.

# Chapter 6

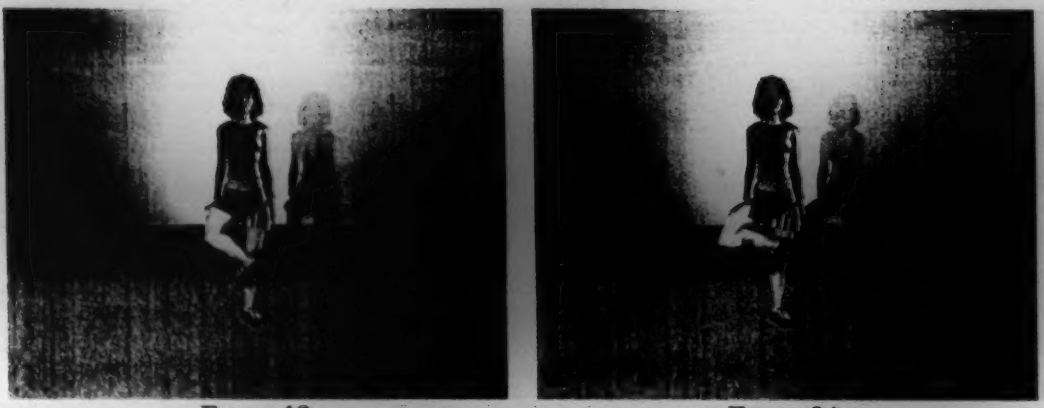
# Results and Evaluation

## 6.1 Results

The Celtic system is comprised of many primitive movements and more complex routines. Primitive movements are not necessarily as simple as lifting a leg or stomping a foot. More complicated primitives exist, such as rotating and bending a leg so that it forms the Celtic "Cut" movement, or jumping in the air and bending the knees on downward impact. One of the main purposes of the Celtic system is to demonstrate that any type of primitive movement can be combined with other primitives to create an interesting sequence of motion. The primitives themselves must look good in order for the viewer's interest to be captured. The results in Figures 6.1 and 6.2 display three primitive movements of different difficulty levels. Figure 6.1 shows the "Cut" primitive movement, which involves both translation and rotation of the leg in a certain way in order to obtain the desired look. This is one of the more complicated primitives because it requires fairly precise rotation and position values. The images in Figure 6.2 present two primitives: "ClickHeelsIn" and "ClickHeelsOut". Unlike the Cut primitive, these movements are fairly simple in their motion, only requiring the rotation of the heels towards and away from each other.

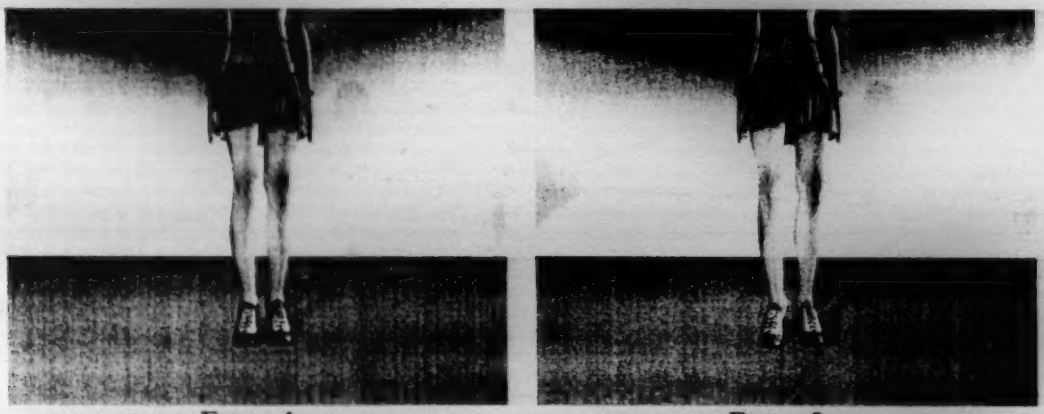
Frame 1

Frame 7

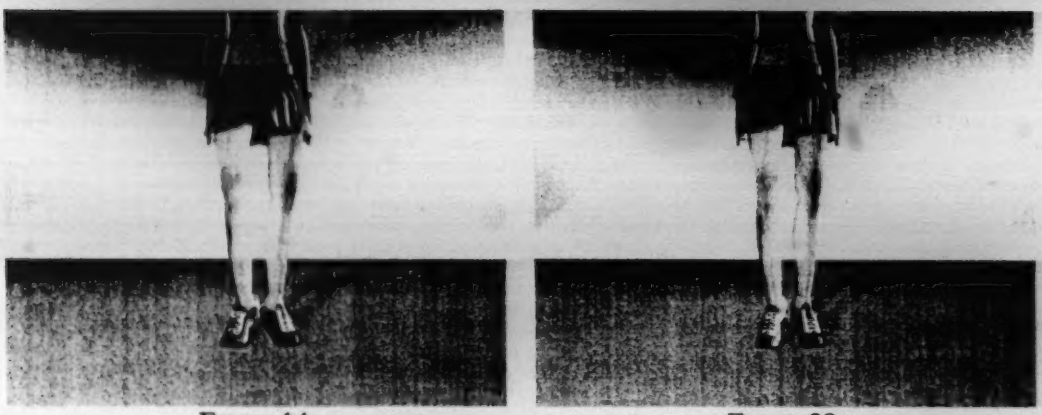


Frame 13 Frame 24

Figure 6.1: Results from the "Cut" primitive movement.



Frame 1 Frame 8



Frame 14 Frame 29

Frame 40

Figure 6.2: Results from the "ClickHeelsIn" (frames 1,8 and 14) and "ClickHeelsOut" (frames 29 and 40) primitive movements.

The Celtic system's routines, whether built-in or user-contributed, generally require the use of several primitive movements. The "FrontClickJump" routine, presented in Figure 6.3, is one of the most interesting routines, and yet it only uses three primitive movements: "LiftLeg," "Hop" and "DropLeg". It is an excellent example of how two fairly simple movements can be combined to create an exciting motion sequence.

Frame 1

Frame 24

Frame 33

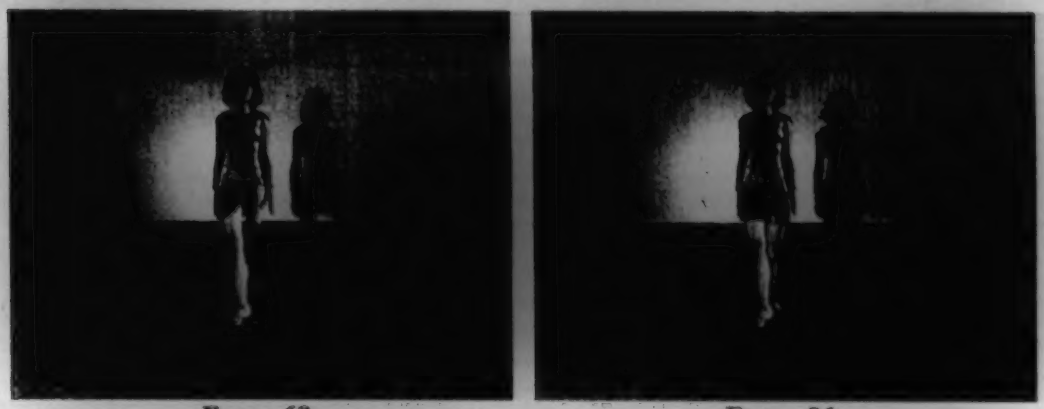
Frame 41

Frame 46

Frame 50

Frame 54

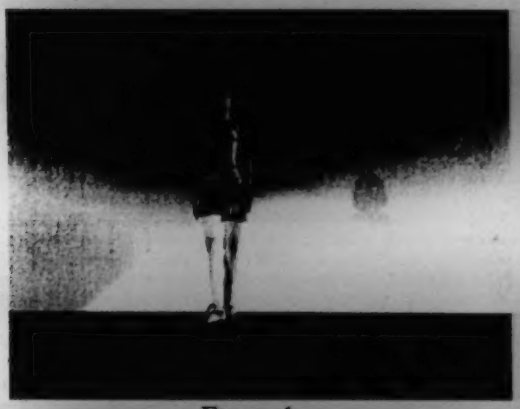
Frame 62



Frame 69 Frame 86

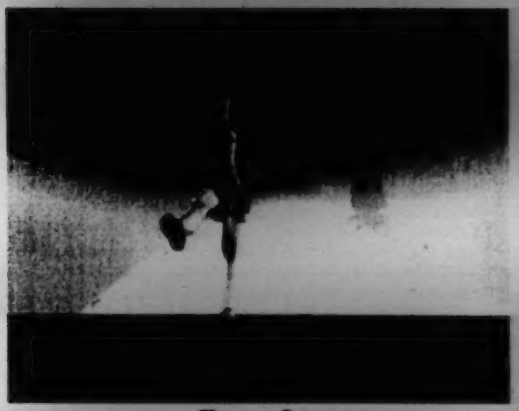
Figure 6.3: Sequential images displaying the different positions involved in the "FrontClickJump" Celtic routine.

The script file provides control over many key aspects of the animation, including the number of characters involved in the scene, the movements performed by each character and the timing of the movements. As far as we know, these features are unique to this system. Figure 6.3 demonstrates the use of the timing features. The character starts the movement in the second image by lifting her leg. In the fourth image, or ¼ of a beat later, she begins the hop motion. Both primitives end at the same time in the last frame. The corresponding script file uses a set of brackets and two rest symbols to create this motion. Each rest symbol represents a delay of ⅛ of a beat, so the hop movement could be started earlier by removing one of the rests or later by adding another rest. The timing features promote tailoring of user-built routines so that users can meticulously create combinations of movements that are not already included in the system.

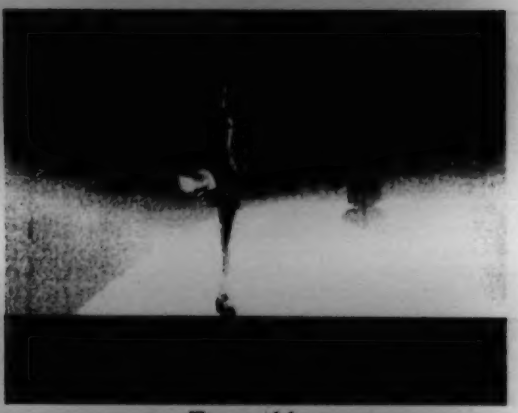


Frame 1

Frame 4



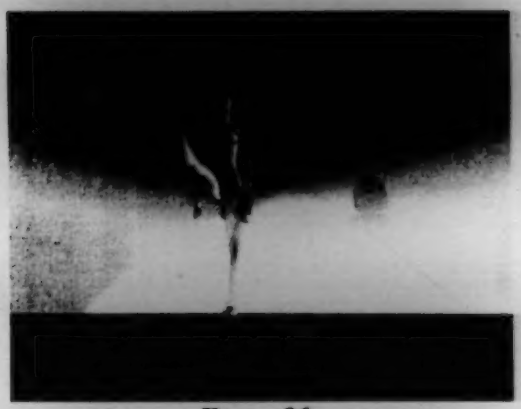
Frame 8



Frame 11

Frame 15

Frame 18



Frame 26

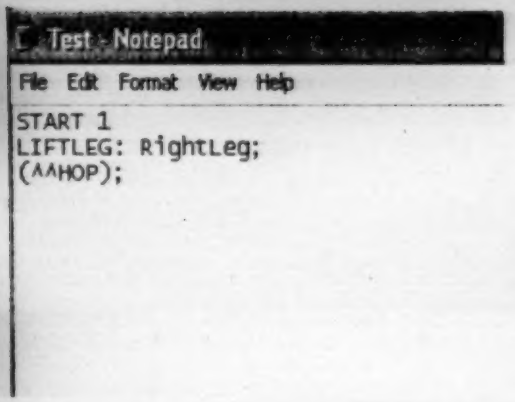



Figure 6.3: These results display one way of using the Celtic system's timing aspects to combine movements. The corresponding script file is shown in the last row on the right.

Multiple characters in a Celtic performance can make the movements more dramatic and interesting for the viewer. The Celtic system supports two types of multiple character movement: synchronized and unsynchronized. Synchronized movement, as shown in Figure 6.5, involves all the characters performing the same movement at the same time. This particular scene involves sixteen characters using the same script file. Figure 6.6 demonstrates unsynchronized movement between six characters. The first and sixth characters are performing a "Sidestep" movement in all the images, while the second and fifth characters are performing a "Shufflehopback" routine in images 1-6 and a "Cut" motion in images 7-11. The third and fourth characters are performing a "Cut" motion in images 1-6 and a "CutBack" motion in images 7-11. Each group of characters is performing at the same time as the other groups but their movements are not the same, resulting in an unsynchronized performance. These results demonstrate how different characters can possess different personalities and yet still fit into the overall presentation. More examples of synchronized and unsynchronized routines can be found in the supplemental video.

Figure 6.4: The system is easily able to accommodate multiple characters in the same scene, as demonstrated in the picture above. Sixteen girls are utilized in this particular performance.

Frame 980

Frame 983

Frame 985

Frame 994

Frame 996

Frame 999

Frame 1001

Frame 1002

Frame 1007

Figure 6.5: Results displaying how the system can use multiple characters and synchronize them all to perform the same motion at the same time. The characters in this scene are performing the "Jumpback" Celtic routine.

Frame 1212

Frame 1214

Frame 1217

Frame 1220

Frame 1233

Frame 1235

Frame 1238

Frame 1244

Frame 1250 Frame 1254

Figure 6.6: Results from six characters performing unsynchronized movement. The characters are split into three groups of two, with each group performing a routine different from the other groups.

The timing values for the animations used in the Evaluation and the animations with multiple characters shown above are displayed in Table 6.1. The values in this table demonstrate how long it takes to build a script file, bake the Inverse Kinematic keys in Maya and render the final result. The time trials were performed on an AMD Athlon 64 3000+ running at 1.81GHz with 2.0GB of RAM. The table gives the user an idea of how long it takes to put together an animation like the ones displayed in the accompanying video. It is important to note that there are values in the Building Script File column that are 0. This is because the script file built for BrownEyedGirl was used for all the other animations without a time value. The reusability of the script file reduced the amount of time needed to create these animations by removing the need to build a script file from scratch. This displays one of the important aspects of the Celtic system. The IK baking time and the rendering time are highly dependent on the number of characters in a scene. The two animations with multiple characters, TwentyFieryNights and Six_Warriors, take more time to bake and render because they involve sixteen and six characters respectively. The number of frames involved in each animation also affects the baking and rendering times. The longer an animation, the higher the values in these columns should be. The number of frames used by each animation is noted in Appendix A.

| Animation | Manually Building the Script File | Baking IK Keys (minutes) | Rendering (hours) |
|---|---|---|---|
| BrownEyedGirl | 30-60 mins | 17 | 13 |
| Eminem | 0 | 7 | 5.5 |
| FieryNights | 0 | 12 | 9 |
| Finale | 0 | 12 | 9 |
| GetItStarted | 0 | 12 | 9 |
| Nutcracker | 0 | 19 | 16 |
| Six_Warriors | 6-9 hrs | 80 | 15-18 |
| Twenty_FieryNights | 1-1.5 hrs | 4-6 hrs | 45 |
| Warriors | 0 | 9 | 7 |
| WideOpenSpaces | 0 | 14 | 12 |

Table 6.1: Values that represent the time taken to build the script file, bake IK keys in Maya and render the entire animation for each song.

## 6.2 Evaluation

The evaluation of a piece of music or a dance performance is generally subjective and extremely dependent on the preferences of the listener or viewer. This makes it exceptionally difficult to quantitatively determine if an animation is good or not. A qualitative evaluation was designed to assess the success of the Celtic system. There are two objectives in performing this evaluation. The first is to determine if the approach taken by the Celtic system is successful in creating appealing animations. The second is to establish if changing the music can also create appealing animations.

The evaluation involves 3 groups of 6 users per group. Each group represents a different user background. The first group incorporates users with dancing experience. These users apply their knowledge of movement to determine if an animation is good or not. The second group includes users with computer programming experience. This group of users has a technical background and will view the animations less artistically than the previous group. They will be able to focus on how well the parts fit together rather than concentrating on how accurate the movements are. The third group incorporates users with neither dancing nor programming experience. These users can

view the animations without any previous prejudices or expectations and are representative of an inexperienced user who may find the system useful.

The evaluation involves 8 animation videos with a single dancer in each. One of our objectives is to determine how different music affects the end result, so a different piece of music is used for each animation. The music types used include celtic, hip-hop, rap, rock, country and classical. The tempos range from 67 bpm to 171 bpm. The evaluator is asked to specify for each animation whether or not she liked the animation. The answer choices are a simple "yes" or "no." She is then asked to states reasons for her answer. The reasons can give us a good idea of how a user's background affects her opinion. The evaluation document requests that the user form an opinion based solely on the merits of a single animation, without comparison to other animations. The evaluation concentrates on determining how successful our approach is by observing how the changing system parameters affect the user's opinion. The evaluation form provided to each user is found in Appendix B.

### 6.2.1 Evaluation Results

The overall results of the evaluation are found in Table 6.2. These results are based on all 18 of the people involved in assessing the 8 animations. The two animations with the highest number of 'yes' answers are both animations using Celtic music. The FieryNights animation was found appealing by 94% of the evaluators, while the Warriors animation was appreciated by 89% of the evaluators. It is interesting to note that the animations with the highest tempo (Eminem at 171 bpm) and the lowest tempo (Nutcracker at 67 bpm) are the animations found the least appealing by the majority of evaluators. The Eminem animation was only enjoyed by 50% of the evaluators, while the Nutcracker animation was liked by only 39%. These songs, however, also belong to musical types that do not typically suit dancing. Both rap and classical are difficult styles for an average person to dance to, so it makes sense that most people would feel that the dancing does not suit the music. The majority of respondents enjoy the remaining four animations, all of which correspond to music types that are traditionally easy to dance to. GetItStarted and WideOpenSpaces were appealing to 78% of evaluators, 72% of participants enjoyed the Finale animation, while BrownEyedGirl was appreciated by 65%

of those involved. It is important to note that due to the small sample size, in many cases a single vote separates the success of one animation over another. For example, a single vote separates Finale, at 13-5, from the GetItStarted and WideOpenSpaces animations, both at 14-4. A single vote also separates BrownEyedGirl at 12-6 from the Finale sequence. A larger sample size is necessary in order to get a true sense of which animations are considered most appealing.

| Animation | Number of 'yes' responses | Number of 'no' responses | Percentage of people who liked the animation |
|---|---|---|---|
| BrownEyedGirl | 12 | 6 | 67% |
| Eminem | 9 | 9 | 50% |
| FieryNights | 17 | 1 | 94% |
| Finale | 13 | 5 | 72% |
| GetItStarted | 14 | 4 | 78% |
| Nutcracker | 7 | 11 | 39% |
| Warriors | 16 | 2 | 89% |
| WideOpenSpaces | 14 | 4 | 78% |

Table 6.2: Overall results of the evaluation, taking into account the responses of all 18 people involved in the assessment of the animations.

The results from Table 6.2 have been divided based on their respective evaluator groupings. Several animations exist where all members of a group have found the result appealing. Participants with previous dancing experience enjoy Warriors best, with FieryNights and BrownEyedGirl tied for second. Those with computer programming experience enjoy FieryNights, GetItStarted and WideOpenSpaces the most of all the animations. Evaluators with no experience like FieryNights and Warriors the best, with Finale a close second. It is interesting to note that the animations liked best by the programming group all fall within the tempo range of 90-110 bpm. The participants with no experience overwhelmingly enjoy the animations with Celtic style music the most.

The group of dancers also seem to enjoy the animations with Celtic style, as two of the top three animations were paired with Celtic music.

The computer programming group is the only group where the majority of participants disliked an animation. 5 out of 6 evaluators disliked both the Eminem and Nutcracker animations. Interestingly enough, these animations represent the fastest and the slowest tempos of all the songs. The evaluators in the group with no experience in dancing or computer programming were undecided as to whether they enjoyed BrownEyedGirl and the Nutcracker, as the responses were split evenly between yes and no. These animations correspond to the two slowest songs of the group. The group of dancers were also split evenly with respect to the Nutcracker animation.

| Animation | Dancing Experience | | Computer Programming Experience | | Neither | |
|---|---|---|---|---|---|---|
| | Yes | No | Yes | No | Yes | No |
| BrownEyedGirl | 5 | 1 | 4 | 2 | 3 | 3 |
| Eminem | 4 | 2 | 1 | 5 | 4 | 2 |
| FieryNights | 5 | 1 | 6 | 0 | 6 | 0 |
| Finale | 4 | 2 | 4 | 2 | 5 | 1 |
| GetItStarted | 4 | 2 | 6 | 0 | 4 | 2 |
| Nutcracker | 3 | 3 | 1 | 5 | 3 | 3 |
| Warriors | 6 | 0 | 4 | 2 | 6 | 0 |
| WideOpenSpaces | 4 | 2 | 6 | 0 | 4 | 2 |

Table 6.3: Results of the evaluation split up by group into evaluators with dancing experience, evaluators with computer programming experience and evaluators with experience in neither.

### 6.2.2 Discussion

According to the results displayed in Table 6.3, each group of participants views the animations in a different way and has a different opinion as to what constitutes an appealing animation. Those with computer programming experience appear to enjoy the

animations within a certain tempo range best, while evaluators in the groups with a dancing background and with no experience appreciate the animations paired with Celtic music. Both the tempo and musical type of a song have emerged as key factors in determining how appealing an animation is to a viewer. Background experience, however, also seems to make an impact on how a person views an animation.

Each participant was asked to give reasoning behind her response for each animation. The responses of those in the dancing group focus on the motion itself and how realistic it appears. They enjoy animations where the motion is believable, which correlates well to their background experience. Most members of the computer programming group discuss the synchronization of the moves to the music. Their responses focus on how well the movements match the music and they all looked for association between the beats in the music and the timing of the motion. Based on their responses, these participants appear to find an animation appealing if it correlates well to the music. It is our belief that this is due to their logical background and their ability to examine how well pieces fit together. The group with no related background experience viewed the animations in a different way than the previous two groups. Rather than focusing on the motion or the timing, they were able to examine the animation as a whole and base their opinions on the overall look of the result. Many responses discussed background features and camera angles, which are important aspects of a performance that can capture and hold a viewer's attention. Participants in this group made mention of the motion and how well it suited the music, but their comments generally pertained to the complete appearance of the animation.

The objectives of this qualitative evaluation were to first determine if our Celtic system is successful in building appealing animations, and secondly to establish if altering the music can change whether an animation is interesting or not. The results of the evaluation show that our objectives have been met. The majority of participants found most of the animations appealing, especially those that used Celtic music with Celtic movement. The results also led to the observation that the style and tempo of the corresponding music can change the attractiveness of the final animation. This leads us to believe that the system discussed in this thesis is successful in achieving its goals and is therefore a worthwhile project.

# Chapter 7

# Conclusion

## 7.1 Contributions

This thesis presents a new music-driven character animation system that supports data-driven mappings of musical features to movements. The system helps users of all experience levels to produce appealing animations based on input music of any type and primitive dance moves and routines. This animation system has progressed through two levels of design and implementation: the Hip-Hop system and the Celtic system. The Hip-Hop system was unsuccessful in generating interesting animations, but its concepts paved the way for an improved character animation system involving Celtic dance. The Celtic system was built on overcoming the shortcomings of the Hip-Hop system, incorporating a better script file set-up, easier integration of multiple characters and more primitive movements. The Celtic system achieves the goals that the Hip-Hop system could not reach: a user-friendly system that creates exciting animations by combining primitive movements into complex motion.

One of the major contributions of this work to the area of character animation is its ability to build a motion sequence directly from extracted musical features. Unlike synchronization-based methods that simply alter an existing animation's timing in accordance to the musical beat, this system creates movements based on the musical beats and dynamics. The movements can easily change to reflect the mood and timing of the music, a feature that is not possible in systems similar to ours.

Another feature that is not supported in other systems is the ability to control multiple characters with different personalities in an animation. The user can build and easily integrate a troupe of dancers into the system. The dancers are not limited to performing the same movements, as the Celtic system is set up so that each character can use its own script file. Synchronization between characters is encouraged, but individuality makes the animation less mundane.

Our system is designed to be flexible for both the user and the animator. The system is set up to support extra primitive movements, as well as more dance types than just Celtic. The addition of other types of movements will encourage experimentation between dance structures, allowing a choreographer to easily mix moves from across different dance categories. Flexibility for the user is provided through both the script file and the musical input. Any type of music with noticeable beats can be used by the system to generate a specifically tailored animation that expresses the music. The script file gives the user a high level of control over the final animation and results that reflect her style and preference.

## 7.2 Future Work

One of the major shortcomings of the system is the occasional inaccuracy of the beat detection algorithm. In some cases, it takes a large amount of manual tweaking to retrieve the correct beat onsets, a problem that is rectified by designing a completely automatic algorithm that can determine beat positions without user interference. Since the musical experience level of the user will vary, it is improbable that she will be able to choose the correct parameters without much experimentation. Future work in the musical analysis section will include a better beat detection algorithm and the extraction of more musical features, including note pitch and melody. Mapping more of the important musical features to the movements will result in an animation that more truthfully represents the music. The tempo detection algorithm will also be improved to include the ability to track tempo changes over the duration of a song. This addition will improve the beat detection algorithm by making it more robust to songs with multiple mood changes.

Currently the system's primitive movements are implemented according to the proportions of the character displayed in the Results section. The script file will easily accept characters of varying heights and proportions, but the movements will not map to these characters correctly. Further investigation into mapping movements between characters of different proportions will increase the flexibility of the system. Another option for dealing with this problem is the implementation of collision detection. Collision detection will prevent interpenetration of limbs of any character by taking measures to move the body part around the point rather than through it. This technique

would remove the need for specialized mapping for characters of different proportions, but it may be more difficult and time-consuming to implement.

In order to faithfully represent Celtic dance, more primitives and routines need to be built into the system. The addition of extra movements will give the user more choices when designing a dance and produce a more accurate depiction of real Celtic performances. The system does not need to be limited to Celtic dance, however. Different types of dances can be added to future versions in order to increase the scope of the system and encourage experimentation between styles. Ballroom dances such as the Waltz or culture-based dances such as the Spanish Flamenco are among the possible dance types that could be incorporated into the Celtic system.

Lastly, the ability to randomly generate sections of a dance, or even an entire dance, automatically is a concept that should be included in the Celtic system. A simplified method was incorporated into the Hip-Hop system but it was never integrated in the Celtic system due to a lack of time. This function can be used to demonstrate the system to new users or fill in movements when a user has run out of ideas. It would increase the flexibility of the system and provide extra help for users with little experience or only a short amount of time.

# References


[1] Alankus, G., Bayazit, A.A., and Bayazit, B. Automated Motion Synthesis for Dancing Characters. *Computer Animation and Virtual Worlds 16*, 3-4 (2005), 259-271.

[2] Arikan, O. and Forsyth, D.A. Interactive Motion Generation from Examples. *ACM Transactions on Graphics 21*, 3 (2002), 483-490.

[3] Brand, M. Voice Puppetry. *International Conference on Computer Graphics and Interactive Techniques*, 21-28, 1999.

[4] Brand, M. and Shan, K. Voice-driven Animation. Technical Report TR1998-020, Mitsubishi Electric Research Laboratories, Cambridge, Massachusetts, 1998. Also appears, Workshop on Perceptual User Interfaces, San Francisco, California, November 1998.

[5] Calvert, T., Wilke, L., Ryman, R., and Fox, I. Applications of Computers to Dance. *IEEE Computer Graphics and Applications* (March/April 2005), 6-12.

[6] Cardle, M., Barthe, L., Brooks, S. and Robinson, P. Music-Driven Motion Editing: Local Motion Transformation Guided by Music Analysis. In *Proceedings of Eurographics UK*, 38-44, 2002.

[7] Cao, Y., Tien, W.C., Faloutsos, P., and Pighin, F. Expressive Speech-Driven Facial Animation. *ACM Transactions on Graphics 24*, 4 (October 2005), 1283-1302.

[8] Dance Forms 1.0. http://www.charactermotion.com/danceforms/. Viewed on August 30, 2006.

[9] Dance Notation Bureau. http://www.dancenotation.org/lnbasics/frame0.html. Viewed on June 21, 2006.

[10] Dixon, S. On the Analysis of Musical Expression in Audio Signals. *Storage and Retrieval for Media Databases 5021* (2003), 122-132.

[11] Dunne, C. *Celtic Feet*. ISBN: 156938147X. Acorn Media, 1996.

[12] Faloutsos, P., van de Panne, M., and Terzopoulos, D. The Virtual St Stuntman: Dynamic Characters with a Repertoire of Autonomous Motor Skills. *Computers & Graphics 25*, 7 (2001), 933-953.

[13] Faloutsos, P., van de Panne, M., and Terzopoulos, D. Composable Controllers for Physics-Based Character Animation. In *Proceedings of SIGGRAPH*, 251-260, 2001.

[14] Fod, A., Mataric, M.J., and Jenkins, O. Automated Derivation of Primitives for Movement Classification. *Autonomous Robots 12*, 1 (January 2002), 39-54.

[15] Fliege, N.J. *Multirate Digital Signal Processing*. John Wiley & Sons, Ltd. Chichester, 1994, 141-143, 256-258.

[16] Gonzalez, R.C. and Wintz, P. *Digital Image Processing*. Addison-Wesley Publishing Company, Massachusetts, 1977, 36-47, 78-82.

[17] Goto, M. An Audio-based Real-time Beat Tracking System for Music With or Without Drum-sounds. *Journal of New Music Research 30*, 2 (June 2001), 159-171.

[18] Goto, M. and Muraoka, Y. A Beat Tracking System for Acoustic Signals of Music. In *Proceedings of ACM Multimedia*, 365-372, 1994.

[19] Goto, M. and Muraoka, Y. Real-time Beat Tracking for Drumless Audio Signals: Chord Change Detection for Musical Decisions. *Speech Communication 27*, 3-4 (April 1999), 311-335.

[20] Harper, R. and Jernigan, M.E. Self-Adjusting Beat Detection and Prediction in Music. In *Proceedings of Acoustics, Speech, and Signal Processing 4* (May 2004), 245-248.

[21] Jan, J. *Discrete Signal Filtering, Analysis and Restoration.* Institution of Electrical Engineers, 2000, 56.

[22] Jensen, K. and Andersen, T.H. Beat Estimation on the Beat. In *IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, 2003.

[23] Jensen, K. and Andersen, T.H. Real-time Beat Estimation using Feature Extraction. In *Proceedings of Computer Music Modeling and Retrieval Symposium*, ser. Lecture Notes in Computer Science, 2003.

[24] Kim, T., Il Park, S., and Shin, S.Y. Rhythmic-Motion Synthesis Based on Motion-Beat Analysis. In *Proceedings of SIGGRAPH*, 392-401, 2003.

[25] Kovar, L. Gleicher, M. and Pighim, F. Motion Graphs. *ACM Transactions on Graphics*, 473-482, 2002.

[26] Laszlo, J., van de Panne M., and Fiume, E. Interactive Control for Physically-based Animation. *ACM Transactions on Graphics*, 201-208, 2000.

[27] Lee, H. and Lee, I. Automatic Synchronization of Background Music and Motion in Computer Animation. In *Proceedings of Eurographics 24*, 3 (2005), 353-362.

[28] Li, Y., Wang., T., and Shum, H.Y. Motion Texture: A Two-Level Statistical Model for Character Motion Synthesis. *ACM Transactions on Graphics*, 465-472, 2002.

[29] Liu, C.K. and Popovic, Z. Synthesis of Complex Dynamic Character Motion from Simple Animations. *ACM Transactions on Graphics*, 408-416, 2002.

[30] Matlab's Wavelet Toolbox: The Discrete Wavelet Transform. http://www.mathworks.com/access/helpdesk/help/toolbox/wavelet/wavelet.html. Viewed on May 31,2005.

[31] Polikar, R. The Wavelet Tutorial Part IV. http://users.rowan.edu/~polikar/WAVELETS/WTpart4.html. Viewed on May 30, 2005.

[32] Safonova, A., Hodgins, J.K., and Pollard, N.S. Synthesizing Physically Realistic Human Motion in Low-Dimensional, Behavior-Specific Spaces. *ACM Transactions on Graphics 23*, 3 (August 2004), 514-521.

[33] Scheirer, E. Tempo and Beat Analysis of Acoustic Musical Signals. *The Journal of the Acoustical Society of America 103*, 1 (January 1998), 588-601.

[34] Shiratori, T., Nakazawa, A., and Ikeuchi, K. Detecting Dance Motion Structure through Music Analysis. In *Proceedings of International Conference on Face and Gesture Recognition*, 857-862, 2004.

[35] Shiratori, T., Nakazawa, A., and Ikeuchi, K. Dancing-to-Music Character Animation. To appear in *Eurographics 25*, 3 (2006).

[36] Sturman, D. Computer Puppetry. *IEEE Computer Graphics and Applications 18*, 1 (January 1998), 38-45.

[37] Taylor, R., Torres, D., and Boulanger, P. Using Music to Interact with a Virtual Character. In *Proceedings of New Interfaces for Musical Expressions*, 220-223, 2005.

[38] Thorne, M., Burke, D., and van de Panne, M. Motion Doodles: An Interface for Sketching Character Motion. *ACM Transactions on Graphics 23*, 3 (2004), 424-431.

[39] Tzanetakis, G. http://www.cs.uvic.ca/~gtzan/work/projects/pastc.html

[40] Tzanetakis, G., Essl, G., and Cook, P. Audio Analysis using the Discrete Wavelet Transform. In *Proceedings WSES International Conference on Acoustics and Music: Theory and Applications*, 2001.

[41] Uhle, C. and Herre, J. Estimation of Tempo, Micro Time and Time Signature from Percussive Music. In *Proceedings of the 6th International Conference on Digital Audio Effects*, 2003.

[42] Wilke, L., Calvert, T., Ryman, R., and Fox I. From Dance Notation to Human Animation: The LabanDancer Project: Motion Capture and Retrieval. *Computer Animation and Virtual Worlds, 16*, 3-4 (July 2005), 201-211.

[43] Woch, A. and Plamondon, R. Using the Framework of the Kinematic Theory for the Definition of a Movement Primitive. *Motor Control 8*, 4 (October 2004), 547-557.

# Appendix A

# Animation Parameters

## A.1 Common Parameters

The parameters common to all the animations mentioned in this thesis are listed as follows.

SIZE.....640x480
RESOLUTION.....72 pixels/inch
FRAME RATE.....48 fps
FORMAT OF RENDERING.....JPEG images
RENDER SETTINGS.....Production Quality
anti-aliasing
RENDERING SOFTWARE.....Maya and RenderPal
MOVIE GENERATION PROGRAM USED.....VirtualDub
VIEWING PROGRAM USED.....Windows [redacted] Player
or Winamp

## A.2 Altered Parameters

The parameters that are different between all the animations mentioned in this thesis are listed in the table below.

| Animation (.avi) | Beat Detection Threshold | Secondary Script File (.txt) | Maya model file (.mb) | Total number of frames |
|---|---|---|---|---|
| BrownEyedGirl | 76% | singleChar | BAKED_BEGSingle | 4050 |
| Eminem | 90% | singleChar | BAKED_ EminemSingle | 1750 |
| FieryNights | 80% | singleChar | BAKED_ FieryNightsSingle | 2850 |
| Finale | 90% | singleChar | BAKED_FinaleSingle | 2880 |
| GetItStarted | | singleChar | BAKED_GISSingle | 2850 |
| Nutcracker | 90% | singleChar | BAKED_ NutcrackerSingle | 4350 |
| Six_Warriors | 84% | warriorsLeft_1<br>warriorsLeft_2<br>warriorsLeft_3<br>warriorsRight_1<br>warriorsRight_2<br>warriorsRight_3 | masha_Troupe | 2730 |
| Twenty_FieryNights | 80% | fieryGroup | masha_TwentyChars | 4200 |
| Warriors | 84% | singleChar | BAKED_WarriorsS | 2250 |
| WideOpenSpaces | 90% | singleChar | BAKED_WOSSingle | 3300 |

Table A.1: Lists the parameters that change for each animation.

# Appendix B

## Evaluation Form

Please choose the option(s) that best describe you:

☐ Dancing ☐ Computer Programming ☐ Neither

**Instructions:**

Give your opinion on whether or not you like the animation by clicking the appropriate option button. Your opinion should be solely based on an individual animation's merit and not with respect to whether you like it more or less than other animations in the group. After each response, please give a reason for your answer.

| **Animation** | **Response** | |
|---|---|---|
| BrownEyedGirl.avi<br>*Reason:* | ☐ Yes | ☐ No |
| Eminem.avi<br>*Reason:* | ☐ Yes | ☐ No |
| FieryNights.avi<br>*Reason:* | ☐ Yes | ☐ No |
| Finale.avi<br>*Reason:* | ☐ Yes | ☐ No |
| GetItStarted.avi<br>*Reason:* | ☐ Yes | ☐ No |
| Nutcracker.avi<br>*Reason:* | ☐ Yes | ☐ No |
| Warriors.avi<br>*Reason:* | ☐ Yes | ☐ No |

WideOpenSpaces.avi ☐ Yes ☐ No

*Reason:*

Please e-mail the completed form to sauer@cs.ualberta.ca. Your participation is greatly appreciated.

# Appendix C

## Script Files

### C.1 Primary and Secondary Scripts for Evaluation animations

**Primary Script**

```
CHARACTER 1
 LEFTLEG: LeftLegCtrl,
 RIGHTLEG: RightLegCtrl,
 UPPERBODY: upperBodyGroup,
 LOCATOR: Locator,
 DYNAMICS: on;

MAPPING
 CHARACTER 1: SingleChar,
 HOPSTEP: HopStep,
 SHUFFLECLICK: shuffleclick,
 DUALCUT: dualCut;
```

**Secondary Script – "SingleChar.txt"**

```
START 1
SHUFFLEHOPBACK 3;
STEPFORWARD 3;
SHUFFLEHOPBACK 3;
STEPBACK 3;
FRONTCLICKJUMP;
JUMPBACK;
SLIDINGSTEP;
DUALCUT;
DUALCUT;
LIFTLEG: RightLeg;
(^HOP);
DROPLEG: RightLeg;
CUT;
SLIDINGSTEP 2;
KNEEBENDHOP;
JUMPBACK;
SLIDINGSTEP 2;
CUT;
SHUFFLEHOPBACK;
```

```
CUTBACK;
FRONTCLICKJUMP 2;
STEPFORWARD 10;
ZIGZAG 2;
CLICKZIGZAG 2;
TURN: Right;
SLIDINGSTEP 3;
FRONTCLICKJUMP;
JUMPBACK;
STEPBACK 5;
TURN: Left;
KNEEBENDHOP;
SHUFFLEHOPBACK;
KNEEBENDHOP;
SIDESTEP 10;
DUALCUT;
STAMP: RightLeg;
HOP;
STAMP: LeftLeg;
FRONTCLICKJUMP;
STEPFORWARD;
(^HOP);
CUTBACK;
SHUFFLEHOPBACK;
SLIDINGSTEP;
TURN: Left;
STEPFORWARD 3;
HOP;
KNEEBENDHOP;
SHUFFLEHOPBACK;
TURN: Left 3;
STEPFORWARD;
(^HOP);
KNEEBENDHOP;
CUTBACK;
LIFTLEG: LeftLeg;
(^HOP);
DROPLEG: LeftLeg;
STEPBACK;
(^HOP);
SLIDINGSTEP 2;
SHUFFLEHOPBACK;
SLIDINGSTEP 2;
FRONTCLICKJUMP;
STAMP: LeftLeg;
```

## C.2 Primary and Secondary Script Files for SixWarriors.avi

The primary script file below uses terms such as EXPORT_June22:LeftLegCtrl to describe a character's body part. This is a name given to the body part by Maya to distinguish between the different characters. The full name of each body part includes EXPORT_June## due to Maya's import process when importing a character into a scene. The character file imported into the scene is called EXPORT_June 19 and Maya increases the number with each new character that is imported into the scene.

**Primary Script File**

```
CHARACTER 1
  LEFTLEG: EXPORT_June22:LeftLegCtrl,
  RIGHTLEG: EXPORT_June22:RightLegCtrl,
  UPPERBODY: EXPORT_June22:upperBodyGroup,
  LOCATOR: EXPORT_June22:Locator,
  DYNAMICS: off;
CHARACTER 2
  LEFTLEG: EXPORT_June19:LeftLegCtrl,
  RIGHTLEG: EXPORT_June19:RightLegCtrl,
  UPPERBODY: EXPORT_June19:upperBodyGroup,
  LOCATOR: EXPORT_June19:Locator,
  DYNAMICS: off;
CHARACTER 3
  LEFTLEG: EXPORT_June20:LeftLegCtrl,
  RIGHTLEG: EXPORT_June20:RightLegCtrl,
  UPPERBODY: EXPORT_June20:upperBodyGroup,
  LOCATOR: EXPORT_June20:Locator,
  DYNAMICS: off;
CHARACTER 4
  LEFTLEG: EXPORT_June21:LeftLegCtrl,
  RIGHTLEG: EXPORT_June21:RightLegCtrl,
  UPPERBODY: EXPORT_June21:upperBodyGroup,
  LOCATOR: EXPORT_June21:Locator,
  DYNAMICS: off;
CHARACTER 5
  LEFTLEG: EXPORT_June23:LeftLegCtrl,
  RIGHTLEG: EXPORT_June23:RightLegCtrl,
  UPPERBODY: EXPORT_June23:upperBodyGroup,
  LOCATOR: EXPORT_June23:Locator,
  DYNAMICS: off;
 CHARACTER 6
  LEFTLEG: EXPORT_June24:LeftLegCtrl,
  RIGHTLEG: EXPORT_June24:RightLegCtrl,
  UPPERBODY: EXPORT_June24:upperBodyGroup,
  LOCATOR: EXPORT_June24:Locator,
```

DYNAMICS: off;

MAPPING
CHARACTER 1: warriorsLeft_1,
CHARACTER 2: warriorsRight_1,
CHARACTER 3: warriorsLeft_2,
CHARACTER 4: warriorsRight_2,
CHARACTER 5: warriorsLeft_3,
CHARACTER 6: warriorsRight_3,
SHUFFLECLICK: shuffleclick,
**DUALCUT: dualCut;**

**Secondary Script Files**

| warriorsLeft_1.txt | warriorsLeft_2.txt | warriorsLeft_3.txt |
|---|---|---|
| START 1 | START 1 | START 1 |
| WAIT 3; | WAIT 3; | WAIT 3; |
| SIDESTEP 10; | SIDESTEP 10; | SIDESTEP 10; |
| STEPFORWARD 10; | STEPFORWARD 10; | STEPFORWARD 10; |
| SHUFFLEHOPBACK 4; | SHUFFLEHOPBACK 4; | SHUFFLEHOPBACK 4; |
| LIFTLEG: RightLeg; | LIFTLEG: RightLeg; | LIFTLEG: RightLeg; |
| (^HOP); | (^HOP); | (^HOP); |
| DROPLEG: RightLeg; | DROPLEG: RightLeg; | DROPLEG: RightLeg; |
| (^HOP); | (^HOP); | (^HOP); |
| STEPFORWARD 4; | STEPFORWARD 4; | STEPFORWARD 4; |
| SIDESTEP 5; | SIDESTEP 5; | SIDESTEP 5; |
| JUMPBACK; | JUMPBACK; | JUMPBACK; |
| SHUFFLEHOPBACK 5; | SHUFFLEHOPBACK 5; | SHUFFLEHOPBACK 5; |
| KNEEBENDHOP; | KNEEBENDHOP; | KNEEBENDHOP; |
| FRONTCLICKJUMP; | FRONTCLICKJUMP; | FRONTCLICKJUMP; |
| WAIT 2; | JUMPBACK; | WAIT 2; |
| SHUFFLEHOPBACK 8; | STEPFORWARD 8; | STEPBACK 24; |
| DUALCUT; | SIDESTEP 3; | SIDESTEP 3; |
| CUT; | SHUFFLEHOPBACK 4; | WAIT 2; |
| WAIT 2; | DUALCUT; | ZIGZAG 3; |
| ZIGZAG 3; | JUMPBACK; | LIFTLEG: RightLeg; |
| LIFTLEG: RightLeg; | ZIGZAG 3; | (^HOP); |
| (^HOP); | LIFTLEG: RightLeg; | DROPLEG: RightLeg; |
| DROPLEG: RightLeg; | (^HOP); | STEPFORWARD; |
| STEPFORWARD; | DROPLEG: RightLeg; | (^HOP); |
| (^HOP); | STEPFORWARD; | STEPBACK; |
| STEPBACK; | (^HOP); | (^HOP); |
| (^HOP); | STEPBACK; | HOP 2; |
| JUMPBACK; | (^HOP); | SIDESTEP 6; |
| SIDESTEP 3; | JUMPBACK; | HOP 2; |
| JUMPBACK; | SIDESTEP 6; | DUALCUT; |

| | | |
|---|---|---|
| HOP 2;<br>DUALCUT;<br>KNEEBENDHOP;<br>WAIT 2;<br>STAMP: RightLeg;<br>FRONTCLICKJUMP 2;<br>CUT;<br>HOP;<br>STEPBACK 8;<br>CLICKZIGZAG 3;<br>STAMP: LeftLeg;<br>DUALCUT;<br>KNEEBENDHOP;<br>STAMP: RightLeg;<br>SHUFFLEHOPBACK 2;<br>LIFTLEG: RightLeg;<br>(^HOP);<br>DROPLEG: RightLeg;<br>SHUFFLEHOPBACK;<br>LIFTLEG: LeftLeg;<br>(^HOP);<br>DROPLEG: LeftLeg;<br>HOP;<br>STAMP: RightLeg;<br>WAIT 4;<br>KNEEBENDHOP;<br>SHUFFLEHOPBACK;<br>KNEEBENDHOP;<br>CUT;<br>WAIT 4;<br>TURN: Right;<br>SHUFFLEHOPBACK;<br>CLICKZIGZAG 2;<br>HOP;<br>ZIGZAG 2;<br>HOP;<br>TURN: Left;<br>FRONTCLICKJUMP;<br>DUALCUT;<br>TURN: Left 4;<br>STAMP: RightLeg; | JUMPBACK;<br>DUALCUT;<br>WAIT;<br>STAMP: RightLeg;<br>FRONTCLICKJUMP 2;<br>CUT;<br>HOP;<br>STEPBACK 16;<br>DUALCUT;<br>KNEEBENDHOP;<br>STAMP: RightLeg;<br>SHUFFLEHOPBACK 2;<br>LIFTLEG: RightLeg;<br>(^HOP);<br>DROPLEG: RightLeg;<br>SHUFFLEHOPBACK;<br>LIFTLEG: LeftLeg;<br>(^HOP);<br>DROPLEG: LeftLeg;<br>HOP;<br>STAMP: RightLeg;<br>STEPFORWARD 4;<br>KNEEBENDHOP;<br>SHUFFLEHOPBACK;<br>KNEEBENDHOP;<br>CUT;<br>STEPBACK 4;<br>TURN: Left;<br>SHUFFLEHOPBACK;<br>CLICKZIGZAG 2;<br>HOP;<br>ZIGZAG 2;<br>HOP;<br>TURN: Right;<br>FRONTCLICKJUMP;<br>DUALCUT;<br>TURN: Left 4;<br>STAMP: RightLeg; | WAIT;<br>STAMP: RightLeg;<br>FRONTCLICKJUMP 2;<br>CUT;<br>HOP;<br>STEPFORWARD 16;<br>DUALCUT;<br>KNEEBENDHOP;<br>STAMP: RightLeg;<br>SHUFFLEHOPBACK 2;<br>LIFTLEG: RightLeg;<br>(^HOP);<br>DROPLEG: RightLeg;<br>SHUFFLEHOPBACK;<br>LIFTLEG: LeftLeg;<br>(^HOP);<br>DROPLEG: LeftLeg;<br>HOP;<br>STAMP: RightLeg;<br>STEPFORWARD 4;<br>KNEEBENDHOP;<br>SHUFFLEHOPBACK;<br>KNEEBENDHOP;<br>CUT;<br>STEPBACK 4;<br>TURN: Left;<br>SHUFFLEHOPBACK;<br>CLICKZIGZAG 2;<br>HOP;<br>ZIGZAG 2;<br>HOP;<br>TURN: Right;<br>FRONTCLICKJUMP;<br>DUALCUT;<br>**TURN: Left 4;**<br>STAMP: RightLeg; |

| *warriorsRight_1* | *warriorsRight_2* | *warriorsRight_3* |
|---|---|---|
| START 1<br>SHUFFLEHOPBACK; | START 1<br>SHUFFLEHOPBACK; | START 1<br>SHUFFLEHOPBACK; |

| | | |
|---|---|---|
| SIDESTEP 10;<br>STEPFORWARD 10;<br>SHUFFLEHOPBACK 4;<br>LIFTLEG: LeftLeg;<br>(^HOP);<br>DROPLEG: LeftLeg;<br>(^HOP);<br>STEPBACK 4;<br>SIDESTEP 5;<br>HOP 2;<br>SHUFFLEHOPBACK 5;<br>KNEEBENDHOP;<br>FRONTCLICKJUMP;<br>WAIT 2;<br>SHUFFLEHOPBACK 8;<br>DUALCUT;<br>CUT;<br>WAIT 2;<br>ZIGZAG 3;<br>LIFTLEG: RightLeg;<br>(^HOP);<br>DROPLEG: RightLeg;<br>STEPFORWARD;<br>(^HOP);<br>STEPBACK;<br>(^HOP);<br>HOP 2;<br>DUALCUT;<br>LIFTLEG: LeftLeg;<br>DROPLEG: LeftLeg;<br>JUMPBACK 2;<br>DUALCUT;<br>KNEEBENDHOP;<br>WAIT 2;<br>STAMP: RightLeg;<br>FRONTCLICKJUMP 2;<br>CUT;<br>HOP;<br>FRONTCLICKJUMP;<br>LIFTLEG: LeftLeg;<br>(^HOP);<br>DROPLEG: LeftLeg;<br>(^HOP);<br>STAMP: LeftLeg;<br>HOP;<br>CLICKZIGZAG 3; | SIDESTEP 10;<br>STEPFORWARD 10;<br>SHUFFLEHOPBACK 4;<br>LIFTLEG: LeftLeg;<br>(^HOP);<br>DROPLEG: LeftLeg;<br>(^HOP);<br>STEPBACK 4;<br>SIDESTEP 5;<br>HOP 2;<br>SHUFFLEHOPBACK 5;<br>KNEEBENDHOP;<br>FRONTCLICKJUMP;<br>WAIT 2;<br>STEPBACK 8;<br>SIDESTEP 3;<br>SHUFFLEHOPBACK 4;<br>DUALCUT;<br>WAIT 2;<br>ZIGZAG 3;<br>LIFTLEG: RightLeg;<br>(^HOP);<br>DROPLEG: RightLeg;<br>STEPFORWARD;<br>(^HOP);<br>STEPBACK;<br>(^HOP);<br>HOP 2;<br>SIDESTEP 3;<br>JUMPBACK 2;<br>DUALCUT;<br>KNEEBENDHOP;<br>WAIT 2;<br>STAMP: RightLeg;<br>FRONTCLICKJUMP 2;<br>CUT;<br>HOP;<br>STEPFORWARD 8;<br>CLICKZIGZAG 3;<br>STAMP: LeftLeg;<br>DUALCUT;<br>KNEEBENDHOP;<br>STAMP: RightLeg;<br>SHUFFLEHOPBACK 2;<br>LIFTLEG: RightLeg;<br>(^HOP); | SIDESTEP 10;<br>STEPFORWARD 10;<br>SHUFFLEHOPBACK 4;<br>LIFTLEG: LeftLeg;<br>(^HOP);<br>DROPLEG: LeftLeg;<br>(^HOP);<br>STEPBACK 4;<br>SIDESTEP 5;<br>HOP 2;<br>SHUFFLEHOPBACK 5;<br>KNEEBENDHOP;<br>FRONTCLICKJUMP;<br>JUMPBACK;<br>STEPFORWARD 24;<br>SIDESTEP 3;<br>JUMPBACK;<br>ZIGZAG 3;<br>LIFTLEG: RightLeg;<br>(^HOP);<br>DROPLEG: RightLeg;<br>STEPFORWARD;<br>(^HOP);<br>STEPBACK;<br>(^HOP);<br>JUMPBACK;<br>SIDESTEP 9;<br>JUMPBACK;<br>FRONTCLICKJUMP 2;<br>CUT;<br>HOP;<br>STEPBACK 24;<br>SHUFFLEHOPBACK 2;<br>LIFTLEG: RightLeg;<br>(^HOP);<br>DROPLEG: RightLeg;<br>SHUFFLEHOPBACK;<br>LIFTLEG: LeftLeg;<br>(^HOP);<br>DROPLEG: LeftLeg;<br>HOP;<br>STAMP: RightLeg;<br>WAIT 4;<br>KNEEBENDHOP;<br>SHUFFLEHOPBACK;<br>KNEEBENDHOP; |

| | | |
|---|---|---|
| STAMP: LeftLeg;<br>DUALCUT;<br>KNEEBENDHOP;<br>STAMP: RightLeg;<br>SHUFFLEHOPBACK 2;<br>LIFTLEG: RightLeg;<br>(^HOP);<br>DROPLEG: RightLeg;<br>SHUFFLEHOPBACK;<br>LIFTLEG: LeftLeg;<br>(^HOP);<br>DROPLEG: LeftLeg;<br>HOP;<br>STAMP: RightLeg;<br>STEPFORWARD 4;<br>KNEEBENDHOP;<br>SHUFFLEHOPBACK;<br>KNEEBENDHOP;<br>CUT;<br>STEPBACK 4;<br>TURN: Left;<br>SHUFFLEHOPBACK;<br>CLICKZIGZAG 2;<br>HOP;<br>ZIGZAG 2;<br>HOP;<br>TURN: Right;<br>FRONTCLICKJUMP;<br>DUALCUT;<br>TURN: Left 4;<br>STAMP: RightLeg; | DROPLEG: RightLeg;<br>SHUFFLEHOPBACK;<br>LIFTLEG: LeftLeg;<br>(^HOP);<br>DROPLEG: LeftLeg;<br>HOP;<br>STAMP: RightLeg;<br>WAIT 4;<br>KNEEBENDHOP;<br>SHUFFLEHOPBACK;<br>KNEEBENDHOP;<br>CUT;<br>WAIT 4;<br>TURN: Right;<br>SHUFFLEHOPBACK;<br>CLICKZIGZAG 2;<br>HOP;<br>ZIGZAG 2;<br>HOP;<br>TURN: Left;<br>FRONTCLICKJUMP;<br>DUALCUT;<br>TURN: Left 4;<br>STAMP: RightLeg; | CUT;<br>WAIT 4;<br>TURN: Right;<br>SHUFFLEHOPBACK;<br>CLICKZIGZAG 2;<br>HOP;<br>ZIGZAG 2;<br>HOP;<br>TURN: Left;<br>FRONTCLICKJUMP;<br>DUALCUT;<br>TURN: Left 4;<br>STAMP: RightLeg; |

## C.3 Primary and Secondary Script Files for TwentyFieryNights.avi

**Primary Script File**

```
CHARACTER 1
  LEFTLEG: LeftLegCtrl,
  RIGHTLEG: RightLegCtrl,
  UPPERBODY: upperBodyGroup,
  LOCATOR: Locator,
  DYNAMICS: off;
CHARACTER 2
  LEFTLEG: EXPORT_MashaLocator3:LeftLegCtrl,
  RIGHTLEG: EXPORT_MashaLocator3:RightLegCtrl,
```

```
  UPPERBODY: EXPORT_MashaLocator3:upperBodyGroup,
  LOCATOR: EXPORT_MashaLocator3:Locator,
  DYNAMICS: off;
CHARACTER 3
  LEFTLEG: EXPORT_MashaLocator4:LeftLegCtrl,
  RIGHTLEG: EXPORT_MashaLocator4:RightLegCtrl,
  UPPERBODY: EXPORT_MashaLocator4:upperBodyGroup,
  LOCATOR: EXPORT_MashaLocator4:Locator,
  DYNAMICS: off;
CHARACTER 4
  LEFTLEG: EXPORT_MashaLocator5:LeftLegCtrl,
  RIGHTLEG: EXPORT_MashaLocator5:RightLegCtrl,
  UPPERBODY: EXPORT_MashaLocator5:upperBodyGroup,
  LOCATOR: EXPORT_MashaLocator5:Locator,
  DYNAMICS: off;
CHARACTER 5
  LEFTLEG: EXPORT_MashaLocator6:LeftLegCtrl,
  RIGHTLEG: EXPORT_MashaLocator6:RightLegCtrl,
  UPPERBODY: EXPORT_MashaLocator6:upperBodyGroup,
  LOCATOR: EXPORT_MashaLocator6:Locator,
  DYNAMICS: off;
CHARACTER 6
  LEFTLEG: EXPORT_MashaLocator7:LeftLegCtrl,
  RIGHTLEG: EXPORT_MashaLocator7:RightLegCtrl,
  UPPERBODY: EXPORT_MashaLocator7:upperBodyGroup,
  LOCATOR: EXPORT_MashaLocator7:Locator,
  DYNAMICS: off;
CHARACTER 7
  LEFTLEG: EXPORT_MashaLocator8:LeftLegCtrl,
  RIGHTLEG: EXPORT_MashaLocator8:RightLegCtrl,
  UPPERBODY: EXPORT_MashaLocator8:upperBodyGroup,
  LOCATOR: EXPORT_MashaLocator8:Locator,
  DYNAMICS: off;
CHARACTER 8
  LEFTLEG: EXPORT_MashaLocator9:LeftLegCtrl,
  RIGHTLEG: EXPORT_MashaLocator9:RightLegCtrl,
  UPPERBODY: EXPORT_MashaLocator9:upperBodyGroup,
  LOCATOR: EXPORT_MashaLocator9:Locator,
  DYNAMICS: off;
CHARACTER 9
  LEFTLEG: EXPORT_MashaLocator10:LeftLegCtrl,
  RIGHTLEG: EXPORT_MashaLocator10:RightLegCtrl,
  UPPERBODY: EXPORT_MashaLocator10:upperBodyGroup,
  LOCATOR: EXPORT_MashaLocator10:Locator,
  DYNAMICS: off;
CHARACTER 10
```

```
 LEFTLEG: EXPORT_MashaLocator11:LeftLegCtrl,
 RIGHTLEG: EXPORT_MashaLocator11:RightLegCtrl,
 UPPERBODY: EXPORT_MashaLocator11:upperBodyGroup,
 LOCATOR: EXPORT_MashaLocator11:Locator,
 DYNAMICS: off;
CHARACTER 11
 LEFTLEG: EXPORT_MashaLocator12:LeftLegCtrl,
 RIGHTLEG: EXPORT_MashaLocator12:RightLegCtrl,
 UPPERBODY: EXPORT_MashaLocator12:upperBodyGroup,
 LOCATOR: EXPORT_MashaLocator12:Locator,
 DYNAMICS: off;
CHARACTER 12
 LEFTLEG: EXPORT_MashaLocator13:LeftLegCtrl,
 RIGHTLEG: EXPORT_MashaLocator13:RightLegCtrl,
 UPPERBODY: EXPORT_MashaLocator13:upperBodyGroup,
 LOCATOR: EXPORT_MashaLocator13:Locator,
 DYNAMICS: off;
CHARACTER 13
 LEFTLEG: EXPORT_MashaLocator14:LeftLegCtrl,
 RIGHTLEG: EXPORT_MashaLocator14:RightLegCtrl,
 UPPERBODY: EXPORT_MashaLocator14:upperBodyGroup,
 LOCATOR: EXPORT_MashaLocator14:Locator,
 DYNAMICS: off;
CHARACTER 14
 LEFTLEG: EXPORT_MashaLocator15:LeftLegCtrl,
 RIGHTLEG: EXPORT_MashaLocator15:RightLegCtrl,
 UPPERBODY: EXPORT_MashaLocator15:upperBodyGroup,
 LOCATOR: EXPORT_MashaLocator15:Locator,
 DYNAMICS: off;
CHARACTER 15
 LEFTLEG: EXPORT_MashaLocator16:LeftLegCtrl,
 RIGHTLEG: EXPORT_MashaLocator16:RightLegCtrl,
 UPPERBODY: EXPORT_MashaLocator16:upperBodyGroup,
 LOCATOR: EXPORT_MashaLocator16:Locator,
 DYNAMICS: off;
CHARACTER 16
 LEFTLEG: EXPORT_MashaLocator17:LeftLegCtrl,
 RIGHTLEG: EXPORT_MashaLocator17:RightLegCtrl,
 UPPERBODY: EXPORT_MashaLocator17:upperBodyGroup,
 LOCATOR: EXPORT_MashaLocator17:Locator,
 DYNAMICS: off;

MAPPING
 CHARACTER 1: fieryGroup,
 CHARACTER 2: fieryGroup,
 CHARACTER 3: fieryGroup,
```

CHARACTER 4: fieryGroup,
CHARACTER 5: fieryGroup,
CHARACTER 6: fieryGroup,
CHARACTER 7: fieryGroup,
CHARACTER 8: fieryGroup,
CHARACTER 9: fieryGroup,
CHARACTER 10: fieryGroup,
CHARACTER 11: fieryGroup,
CHARACTER 12: fieryGroup,
CHARACTER 13: fieryGroup,
CHARACTER 14: fieryGroup,
CHARACTER 15: fieryGroup,
CHARACTER 16: fieryGroup,
SHUFFLECLICK: shuffleclick,
DUALCUT: dualCut;

**Secondary Script File**

START 1
SHUFFLEHOPBACK 3;
CLICKZIGZAG 3;
SHUFFLEHOPBACK 3;
CLICKZIGZAG 3;
SIDESTEP 2;
SHUFFLEHOPBACK;
STAMP: RightLeg;
SIDESTEP 2;
SHUFFLEHOPBACK 3;
KNEEBENDHOP;
SHUFFLEHOPBACK 3;
KNEEBENDHOP;
SLIDINGSTEP 2;
JUMPBACK;
SLIDINGSTEP 2;
SHUFFLECLICK;
FRONTCLICKJUMP;
SHUFFLECLICK;
FRONTCLICKJUMP;
HOP;
STAMP: LeftLeg;
STAMP: LeftLeg;
KNEEBENDHOP;
TURN: Left 2;
STEPFORWARD 4;
TURN: Right 2;

```
DUALCUT;
JUMPBACK;
DUALCUT;
LIFTLEG: RightLeg;
(^HOP);
DROPLEG: RightLeg;
(^HOP);
STAMP: LeftLeg;
STAMP: LeftLeg;
HOP;
SHUFFLEHOPBACK 3;
STAMP: RightLeg;
DUALCUT;
DUALCUT;
HOP;
FRONTCLICKJUMP 2;
SHUFFLECLICK;
SHUFFLECLICK;
SIDESTEP 5;
JUMPBACK;
SHUFFLECLICK;
SHUFFLECLICK;
SIDESTEP 6;
FRONTCLICKJUMP 2;
SLIDINGSTEP;
SHUFFLEHOPBACK;
SLIDINGSTEP;
LIFTLEG: LeftLeg;
(^HOP);
DROPLEG: LeftLeg;
(^HOP);
HOP;
KNEEBENDHOP;
CUTBACK;
DUALCUT;
STAMP: LeftLeg;
STAMP: RightLeg;
HOP;
SHUFFLEHOPBACK 2;
STEPFORWARD;
(^HOP);
STEPBACK;
(^HOP);
JUMPBACK 2;
FRONTCLICKJUMP;
DUALCUT;
```

DUALCUT;
SIDESTEP;
SLIDINGSTEP;
SHUFFLEHOPBACK;
SIDESTEP;
SLIDINGSTEP;
TURN: Left 4;